Reasoning about Selective Strictness

Operational Equivalence, Heaps and Call-by-Need Evaluation, New Inductive Principles

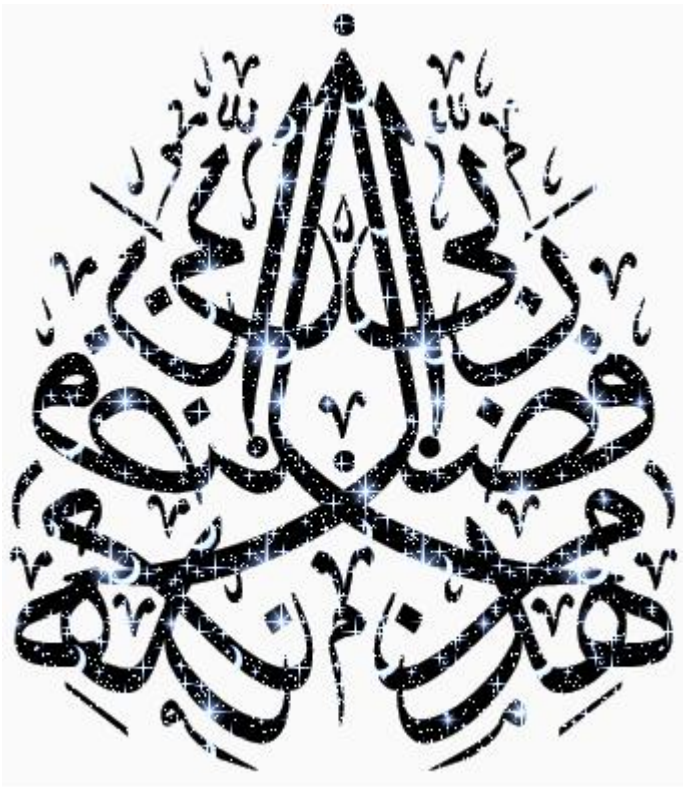
By:
Seyed H. HAERI (Hossein)

Supervisors:
Dr. Murdoch J. Gabbay and Prof. Philippe De Wilde

Submitted for the Degree of
Master of Philosophy
at Heriot-Watt University
on Completion of Research in the
School of Mathematical and Computer Sciences
Aug 2009

بسم الله الرحمن الرحيم

In the name of God, the compassionate, the merciful.



This is of the favour of my Lord.

Abstract

This thesis studies how to prove equivalences between programs in languages with selective strictness, specifically, we use a restricted core lazy functional language with a selective strictness operator seq. We establish some of the first ever equivalences between lazy programs with selective strictness by manipulating operational semantics derivations. Our operational semantics is similar to that used by van Eekelen and De Mol, though we introduce a 'garbage-collecting' rule for (**let**) which turns out to cause expressiveness restrictions. For example, arguably reasonable lazy programs such as $\mathsf{let}\ y = \lambda z.z\ \mathsf{in}\ \lambda x.y$ do not reduce in our operational semantics.

We prove some properties of seq, including associativity, idempotence, and left-commutativity. The proofs use our three notions of program equivalence defined on a big step operational semantics for a core functional language with selective strictness. We also show that, our garbage-collecting (**let**) rule causes the apparently weaker $\approxeq_v$ to coincide with $\approxeq_s$. Left-commutativity is particularly difficult to prove as it is necessary to consider transforms of derivations which make subderivations larger, rather than smaller, and hence induction on derivations or the size of derivations fails.

To prove our results we make some new fundamental observations about our operational semantics and a theory describing a dependency relation between the evaluation of expressions bound to variables in heaps; there follows a notion of bisimilarity on heaps. These notions are general and are not restricted to selective strictness. We consider a measure of complexity of a derivation which does not have to do with its size; this is $\mathit{diff}(\Pi)$. This is deceptively simple, being just the variables in the heap whose bindings are changed by the evaluation. However, using the dependency relation, we can reason by induction on the size of $\mathit{diff}(\Pi)$. It turns out that this quantity can be conveniently 'made smaller', even if subderivations of Π become larger, and thus we obtain a useful inductive quantity.

To the light of my heart
without whom it would have frozen...

Acknowledgements

The student would like to thank the invaluable helps and directions of Dr. Murdoch J. Gabbay (his primary supervisor), as well as Dr. Phil W. Trinder and Prof. Yolanda Ortega-Mallén.

Contents

List of Figures

Chapter 1

Introduction

Introducing limited, or *selective*, strictness into an otherwise lazy language can greatly improve performance. Selective strictness can improve the time or space efficiency of data structures, for example the unboxed arrays of Haskell and Clean improve both time and space performance. Similarly, selective strictness can improve the time or space efficiency of computations. Folklore abounds with stories of solving space leaks by adding sufficient strictness to release large data structures early in the computation. Finally, selective strictness can be used to control evaluation order; this is especially important for parallel evaluation. For instance, the parallel language Eden [60] extends Haskell with mechanisms for eager process creation and eager value communication.

Selective strictness is introduced using a variety of mechanisms including special data structures like unboxed arrays, type annotations, and language constructs. Indeed, many languages offer more than one selective strictness mechanism; both Haskell and Clean offer all three of the mechanisms mentioned above. In this thesis, we focus on one selective strictness language construct, the seq in Haskell, which also corresponds closely to the let! in Clean.

Selective strictness is useful, and some would argue that it is essential, for effective software development in lazy languages. However, it is also true that unconsidered use of strictness can degrade program time or space performance, and even cause a previously terminating program to diverge. Programmers are warned of these hazards, see for example [77, Section 6.2]. So, we need to be able to reason about this powerful but dangerous construct.

1.1 Contributions

The purpose of this thesis is therefore to contribute to the relatively limited body of work on reasoning about selective strictness. The research contributions of the work are as follows:

- We establish some of the first ever equivalences between programs with selective strictness. We do this by manipulating operational semantics derivations, in contrast to earlier related works which use a denotational

approach [40] or a theorems-for-free one [47]. Our operational semantics is similar to that used by van Eekelen and De Mol[105], though we introduce a 'garbage-collecting' rule for let.

This garbage-collection restricts the expressiveness of our system, and hence there are some arguably reasonable lazy programs that do not reduce in our operational semantics. As will be demonstrated in Remark 3.5, an example of such programs is let $y = \lambda z.z$ in $\lambda x.y$. In Section 11.3, we illustrate how, as another example, let $x_1 = \lambda x.x$ in ((let $x_2 = \lambda x.x$ in $\lambda x.x_2$) x_1) does not reduce in our system. Despite such restrictions, and apart from the technical benefits garbage-collection brings to our proofs (in Theorem 8.4 and Corollary 8.6 for example), as discussed in Section 11.3, we still believe that garbage-collection is necessary for (**let**) rules in lazy evaluation. (One can also consult Remark 8.13.) How to retain garbage-collecting (**let**) rules without restricting the expressiveness is future work.

- We prove some elementary properties of seq, including associativity (Theorem 5.1, strengthened in Theorem 10.2) and a weak form of idempotence (Theorem 5.5) in Chapter 5. The proofs use our three notions of program equivalence (Chapter 4) defined on a big step operational semantics for a core functional language with selective strictness (Chapter 3).

  We call these properties 'elementary', not necessarily because they are trivial, but because their proofs are by fairly straightforward transformations on derivations in the operational semantics, and by structural inductions on those derivations.

- We prove left-commutativity and idempotence of seq (Theorems 9.3 and 10.2). We also show that, our garbage-collecting (**let**) rule will cause the apparently weaker $\approx_v$ coincides with $\approx_s$ (Section 8.2).

  Here, the proofs are harder. The underlying reason is that it is necessary to consider transforms of derivations which make subderivations larger, rather than smaller; induction on derivations or the size of derivations fails.

  To prove our results, we make some new fundamental observations about our operational semantics (Chapter 6) and a theory describing a dependency relation between the evaluation of expressions bound to variables in heaps (Chapter 7); there follows a notion of equivalence on heaps: $\approx$ (Chapter 8). These notions are general and are not restricted to selective strictness. However, it happens that, for our specific operational semantics (Definition 3.4), $\approx$ is a bisimilarity relation (Section 8.3). In future work, we hope to apply these notions to reasoning about call-by-need evaluation in general, and to parallel evaluation.

  We consider a measure of complexity of a derivation which does not have to do with its size; this is $\mathit{diff}(\Pi)$ (Definition 6.4). This is deceptively simple, being just the variables in the heap whose bindings are changed by the evaluation. However, using the dependency relation, we can prove three central technical results: Theorems 7.10, 7.11, and 7.12. These results

together allow us to reason by induction on the size of *diff*(Π). It turns out that this quantity can be conveniently 'made smaller', even if subderivations of Π become larger, and thus we obtain a useful inductive quantity. The reader can see how this inductive proof-method is applied, in Theorem 8.4, Lemma 9.1, and Theorem 10.1.

- In the course of proving our results we will come across other more technical theorems, which we need for some specific purpose but which are attractive in themselves. Examples are Determinism (Theorem 3.9), Theorem 4.6, Theorem 6.9, and Theorem 6.23.

For the reader's convenience, we open each chapter with a technical outline of the material covered in it.

1.2 Authorship

Chapters 3 to 10 of this thesis are substantially based on joint work with Drs Gabbay, Trinder and Prof Ortega-Mallén [71]. The author's specific contributions to the research are as follows:

- To contribute to the formulation of [71], and to check many of the proof sketches in it.
- To prove a total of 38 theorems, lemmata, and corollaries. More specifically: The author replaced the proof sketches from [71] with complete proofs for 15 theorems, lemmata, and corollaries. He also proved 8 lemmas for which there was no proof in [71]. And, finally, the above 23 results entailed 15 additional lemmata and corollaries all of which the student proved.
- To formulate the notions of equivalence in Chapter 4.
- To prove the Associativity and Idempotence of seq in Chapter 5.
- To contribute to the development of Analogous Heaps ($\approx$) in Chapter 8, conjecture that it gives rise to a bisimilarity, formulate Definition 8.16 with a hint from Prof Ortega-Mallén, and to prove this conjecture (Section 8.3).
- To contribute in the development of *induction on the size of diff*(Π). This is through suggesting earlier formulations for *diff*(Π) and refuting all the similar earlier strategies for proving Left-commutativity (Theorem 9.3). It is also noteworthy that it was the author who first suggested Left-Commutativity.

1.3 Structure

This thesis is structured as follows: we start by reviewing the related work in Chapter 2. Next, in Chapter 3, we present our operational semantics along with

some of its fundamental properties. We then define our notions of equivalence in Chapter 4 and make a few technical observations about them. Elementary properties of seq are considered in Chapter 5. Based on these, in Chapter 6, we look at the semantics of the systems based on Launchbury's operational semantics. Atomic variables and how they relate to the internals of heaps are considered in Chapter 7. This leads us to the useful notion of analogous heaps which is studied in Chapter 8. Left-commutativity and idempotence of seq are proved in Chapters 9 and 10, respectively. In Chapter 11, we make concluding remarks and consider some possible future research directions.

Chapter 2

Related Work

So-called 'lazy' programming languages like Haskell and Clean in fact implement call-by-need evaluation as first proposed by Wadsworth [114]. This is normal order reduction to weak head normal form refined with subexpression sharing. Therefore, to reason about selective strictness in such languages, we must first formalise normal order evaluation.

Section 2.1 covers work on call-by-need. Ong and Abramsky [73, 2, 3] studied this, but they did not capture sharing. Both [96] and [55] present a big-step semantics for call-by-need. Context-based representations are defined in [10, 7, 62]. We choose to build on Launchbury's big step semantics [55] (Chapter 3). This models a restricted core call-by-need evaluation at a level of abstraction suitable for our needs; it is more abstract than lazy abstract machines, and more concrete than mostly denotational treatments like those of Abramsky and Ong.

Before going to selective strictness, Section 2.2 discusses the change in semantics of seq over time. Section 2.2, furthermore, explains the current diversity of viewpoint between the campaigns in the Haskell community over the semantics of seq versus its other variation pseq.

Section 2.3 reviews the work on selective strictness. In prior work other researchers have addressed the issues raised by selective strictness. Harrison et al. [37] provides a *calculational* approach, in which every detail of how to implement a Haskell interpreter in Haskell itself is explained. Later, [38] uses P-logic [52] to specify and verify properties of programs with selective strictness. In [106], van Eekelen and de Mol extend Launchbury's semantics to selective strictness. Other related works of this group are [105] and [108]. We choose to directly reason on derivation trees of an operational semantics which is a variation of that of [106]. Johann and Voigtländer [47] study free theorems in presence of selective strictness. [48, 110, 111, 97, 98] all extend this work.

In Section 2.4, we consider other related work.

In Sections 2.1–2.4, works are categorised based on the authors. The categories are then ordered chronologically. Each section starts with an explanation on the common features of the reviewed work of the respective section. Section 2.1 ends in a comparison between the work of the section with that of ours.

$$\frac{}{\lambda x.P \Downarrow \lambda x.P}$$

$$\frac{M \Downarrow \lambda x.P \quad P[x := Q] \Downarrow N}{MQ \Downarrow N}$$

Figure 2.1: Operational Semantics of Abramsky and Ong

2.1 Call-by-Need

In this section, we explore the different work which present models or operational semantics for call-by-need. Ariola et al. [10, 7, 62] chose "call-by-need" to only describe calculi — as opposed to evaluation strategies. In their terminology, "lazy evaluation" is considered as an evaluation strategy for call-by-need. Unlike Ariola et al., we will not distinguish between call-by-need and lazy evaluation or even "laziness". Different properties of each system will be discussed. At the end of the section the decision to chose Launchbury's operational semantics[55] is justified.

Abramsky and Ong (1988, 1990, and 1993)

Abramsky and Ong's motivation is to fill a gap between the existing theory at the time of their writing and the common practice of functional programming languages of the same time [2, 73, 3]. At the time the theory was powerful enough to give meaning to λ-terms up to head-normal form (hnf). Whereas many functional programming languages of the time ceased evaluation when the terms are only in weak-head normal form (whnf). This practice is well-motivated in lazy evaluation, for which they offer a powerful theory.

Ong [73] starts by introducing his operational semantics which comes in Figure 2.1. Next, he acknowledges the definition of *applicative bisimilarity* ($\sim^B$) from Abramsky as:

$$M \sim^B N \Leftrightarrow (M \overset{\sqsubset^B}{\sim} N) \wedge (N \overset{\sqsubset^B}{\sim} M) \quad \text{where}$$
$$M \overset{\sqsubset^B}{\sim} N \Leftrightarrow (M \Downarrow \lambda x.P \Rightarrow N \Downarrow \lambda x.Q \wedge \forall R.P[x := R] \overset{\sqsubset^B}{\sim} Q[x := R]).$$

This definition gives rise to the $\lambda\ell \overset{\text{def}}{=} \langle \Lambda^0, \sqsubseteq, = \rangle$ inequational theory where [1]

$$\lambda\ell \vdash M \sqsubseteq N \overset{\text{def}}{=} M \overset{\sqsubset^B}{\sim} N$$
$$\lambda\ell \vdash M = N \overset{\text{def}}{=} M \sim^B N.$$

The rest [73] explores various properties of $\lambda\ell$. For example, Ong shows that $\overset{\sqsubset^B}{\sim}$ is a Morris-style pre-congruence [70]. More notably, he shows that $\lambda\ell$ is not fully abstract with regard to D — the initial solution to the domain equation: $D \cong [D \to D]_\perp$. Several extensions of $\lambda\ell$ are also explored afterwards.

[1] Λ^0 denotes the collection of all closed λ-terms.

The seminal work [2] starts by constructing Abramsky's theory by the study of *Applicative Transition Systems* (ats). He offers a domain equation for ats'es next. This helps him to show a number of important results about their system such as computational adequacy. He also constructs a domain logic and shows how to employ this for the study of the equational theory over programs. His handy vehicle in [2] is his notion of *applicative simulation* and *bisimulation*. This is essentially based on the same operational semantics as Figure 2.1. He shows that this notions coincide with that of traditional "Morris-Style" contextual equivalence [70, 12, 66, 86]. This work is also equipped with a variety of other technical side-products. For example, he shows that $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ is a least element of their simulation, and **YK** is a top element for that. [2]

The final work of Abramsky and Ong on lazy evaluation is [3] where all the proofs of their previous works are thoroughly filled. As shown in Figure 2.1, their work [2, 73, 3] fails to capture sharing — which is crucially important in the context of lazy evaluation. This is the case because in their study there is no *book-keeping* media; that is, either a store or a rule for appropriately updating let-bindings. (As a matter of fact, let-bindings are completely absent in the work of Abramsy and Ong.) An alternate point of view on this (due to [7]) is that: Abramsky and Ong explored the model theoretic properties of the call-by-need calculus Plotkin offered along with his evaluator to λ-abstractions [85].

Purushothaman and Seaman(1992, 1994, and 1996)

The first work of this group [89] presents the first operational semantics for lazy evaluation which does capture sharing. This is for a language called LAZY-PCF+SHAR which extends the PCF syntax with explicit substitutions. Their operational semantics is based on judgements of the form $\ll e, A \gg \rightarrow \ll e', A' \gg$ where e' might be a simple expression or one with several layers of explicit substitutions (to represent let-bindings). Purushothaman and Seaman establish soundness and computational adequacy with respect to a fixed-point semantics for PCF [86]. Whilst proof sketches are provided in this work, the details are left to [88].

In their operational semantics, As are **list**s of bindings of the form $x_i \mapsto e_i$ where $FV(e_i) \subseteq \{x_j\}_{i+1}^{n}$. In addition to the fact that working with lists rather than sets makes mutual recursion problematic, it imposes several other difficulties in working with their system. For example, variable evaluations need an ordered list traversal until one reaches to the actual variable to be evaluated. (This is Var2 and Var1 in Figure 2.2, respectively.) Another example of such difficulties is the necessity of a lemma which proves that the order of certain bindings can be changed under right conditions (Lemma 4.1 in [89]). Finally, this restriction has also unnaturally affected their Rec rule (Figure 2.2): each unfolding of a recursive expression will produce a fresh binding.

On the other hand, their operational semantics is typed. Although they employ the type information to prove certain desirable well-formedness properties of their system, the necessity of typing remains questionable. Testimony to this

[2] One can think of **YK** as the infinite process solving the equation $\psi = \lambda x.\psi$. [2].

$$\text{Var1:}\quad \frac{\ll e, A \gg \rightarrow \ll N, A' \gg}{\ll x, [x : t \mapsto e].A \gg \rightarrow \ll N, [x : t \mapsto N].A' \gg}$$

$$\text{Var2:}\quad \frac{\ll y, A \gg \rightarrow \ll N, A' \gg}{\ll y, [x : t \mapsto e].A \gg \rightarrow \ll N, [x : t \mapsto e].A' \gg} \quad y \neq x$$

$$\text{Appl:}\quad \frac{\ll e_1, A \gg \rightarrow \ll N, A' \gg \quad \ll Ap(N, e_2), A' \gg \rightarrow \ll N', A'' \gg}{\ll e_1\ e_2, A \gg \rightarrow \ll N', A'' \gg}$$

$$\text{Rec:}\quad \frac{\ll \langle e[nx/x], [nx : s \mapsto \mu x : t.e] \rangle, A \gg \rightarrow \ll N, A' \gg}{\ll \mu x : t.e, A \gg \rightarrow \ll N, A' \gg}$$

$$\begin{array}{l} Ap(\lambda x : t.e_0, e) \qquad\qquad = \langle e_0[nx/x], [nx : t \mapsto e] \rangle \\ Ap(\langle N, [x : t \mapsto e_1] \rangle, e) = \langle \langle K, [x : t \mapsto e_1] \rangle, [n : s \mapsto e] \rangle \\ \qquad\qquad\qquad\qquad\qquad \text{where } Ap(N, e) = \langle K, [n : s \mapsto e] \rangle. \end{array}$$

Note: nx denotes a new variable.

Figure 2.2: Var1, Var2, Appl, and Rec of [89]

is that, in most of their proofs, the typing information is fully dismissed. Finally, the fact that, unlike Launchbury [55], they do not normalise expressions prior to evaluation makes their Appl rule considerably complicated. See Figure 2.2.

In their next work [95], Seaman and Purushothaman change a few minor properties of their previous system [89]. As they put it, their motivation is to provide a formalism for lazy evaluation which both captures sharing and is useful for reasonings. Examples of reasoning that they mention include compile-time analysis such as garbage-collection, order of evaluation, and update-in-place [12, 26, 35]. They argue that because the sharing behaviour will not be present in a denotational model a denotational semantics is not a suitable basis for a proof of correctness of such an analysis. With this in mind, they change some aspects of LAZY-PCF+SHAR in the ways that will be explained later.

Unlike their previous work [89], this time, they prove the computational correctness of their semantics by showing that it is equivalent to a call-by-name operational semantics found in [34]. In order to do this, they offer an intermediate semantics in which only all the typing information are stripped. This puts yet more question on the necessity of type information in their operational semantics. They also prove some nice properties of their operational semantics which are present in our system. This includes: Determinism (Thoerem 3.9), Lemma 5.3, Lemma 5.4, and a weaker form of Lemma 3.10.

The judgments of Seaman and Purushothaman in [95] are of the form: $\langle e, A \rangle \downarrow \langle e', A' \rangle$. Below is a discussion on the changes of this operational semantics over [89]. More details on the exact rules comes in Figure 2.3:

$$\{\text{Appl}\}\quad \frac{\langle e_1, A\rangle \downarrow \langle \lambda x : s.e, A'\rangle \quad \langle e[nx/x], [nx : s \mapsto e_2]A'\rangle \downarrow \langle e', A''\rangle}{\langle (e_1\ e_2), A\rangle \downarrow \langle e', A''\rangle}$$

$$\{\text{Rec}\}\quad \frac{\langle e[nx/x], [nx : s \mapsto \mu x : t.e]A\rangle \downarrow \langle e', A'\rangle}{\langle \mu x : t.e, A\rangle \downarrow \langle e', A'\rangle}$$

Figure 2.3: The {Appl} and {Rec} of [95]

1. They remove the explicit substitutions from their syntax; they handle substitutions in their operational semantics only. As they explain explicit substitutions, per se, are not suitable for modelling sharing in its lazy evaluation fashion. Therefore, they decide to dismiss this in their syntax. However, despite the removal of explicit substitutions from their syntax, they still do not add any syntax for let-binding.

2. They update the Appl rule of Figure 2.2 to {Appl} rule of Figure 2.3. Despite the change — which simply accommodates removal of explicit substitutions — not normalising arguments prior to evaluation causes complications in the operational semantics: they still need to augment their environment on the spot. That is replacing A' by $[nx : s \mapsto e_2]A'$ (list concatenation).

3. Similarly, the removal of explicit substitutions from their syntax demands a new rule for recursions (Rec in Figure 2.2). Making substitutions implicit does not remove their need for production of fresh bindings upon every unfolding of a recursive expression. This prevents their semantics from capturing sharing of recursive functions.

The final work of this group [96] is similar to [95], with some minor changes which will be explained later. In [96], Seaman and Purushothaman report that their operational semantics is used as a basis for an analysis called reduction of variables [90] which indicates whether the result of an evaluation is referred to by some variable that could be used later in the program. Like their previous operational semantics, this third semantics also suffers from addition of fresh bindings upon every unfolding of recursion. They suggest two ways to overcome this problem. Regardless of the fact that neither of their suggestions is shown to work, implementing either suggestion would dictate a drastic change to their entire machinery. (See the relevant discussion of Seaman and Purushothaman in Section 2.6 of [96].)

The judgements of the operational semantics offered in [96] are of the form: $\langle e, A\rangle_Z \downarrow \langle v, A'\rangle_{Z'}$ where the subscript is a list of fresh names, and v ranges over values. The major differences between this semantics and their previous ones are on the rules shown in Figure 2.4. Most notably, they merge {Var1} and {Var2} into {Var} to remove the need for list traversal for every variable evaluation.

$$\{\text{Var}\} \quad \frac{\langle e, A\rangle_Z \downarrow \langle v, A'\rangle_{Z'}}{\langle x, A_0[x : t \mapsto e]A\rangle_Z \downarrow \langle v, A_0[x : t \mapsto v]A'\rangle_{Z'}}$$

$$\{\text{Appl}\} \quad \frac{\langle e_1, A\rangle_Z \downarrow \langle \lambda x : s.e, A'\rangle_{z:Z'} \quad \langle e[z/x], [z : s \mapsto e_2]A'\rangle_{Z'} \downarrow \langle v, A''\rangle_{Z''}}{\langle (e_1\ e_2), A\rangle_Z \downarrow \langle v, A''\rangle_{Z''}}$$

$$\{\text{Rec}\} \quad \frac{\langle e[z/x], [z : t \mapsto \mu x : t.e]A\rangle_Z \downarrow \langle v, A'\rangle_{Z'}}{\langle \mu x : t.e\rangle_{z:Z} \downarrow \langle v, A'\rangle_{Z'}}$$

Figure 2.4: The {Var}, {Appl}, and {Rec} of [96]

Launchbury (1993)

Launchbury's work [55] is of an intermediate level of granularity, being at a higher level of abstraction than the variety of lazy abstract machines [49, 76, 27, 53, 99], and lower level than the full-fledged theories [2, 73, 3]. He offers a natural semantics [87, 51] for lazy evaluation that does capture sharing. Because his natural semantics captures a range of commonalities between these machines, it offers a fertile framework for the study of different aspects of behaviours of lazy functional programmes. For example, Launchbury himself speaks of the possibility of expanding his work for taking "Garbage Collection" and "Cost of Computation" into account. Launchbury's work truly captures sharing, and provides proofs for the Soundness, Completeness, and Computational Adequacy of his heap-based model (with respect to the Denotational Semantics that he presents).

Launchbury's big-step operational semantics is illustrated in Figure 2.5. (Note that in his semantics z ranges over values.) What perfectly fits this semantics for our application is that, in addition to a means for lazily evaluating the expressions, it tracks the effects of evaluations in their environments. The medium which utilises this is the notion of **heap**s which play a critical role in our work as well. As analysed earlier, the similar work which provide store-like notions (such as Purushothaman and Seaman [89, 95, 96]) do not completely dovetail into our purposes because they put constraints on the order of bindings.

Ariola et al. (1995, 1997, and 1998)

The first work of this group [10] aims to find a match between the operational semantics of λ-calculus and the actual behaviour of lazy functional languages by the time [10] was written. They argue that the low-level nature of natural semantics such as [89, 55, 95, 96] permits neither a simple explanation of language implementations, nor source-level reasoning about program behaviour. Over the course of time, we now however know that this is not true. For example, the variety of interesting properties in [55], provides an elegant basis for explaining

$$\frac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e} \quad \textit{Lambda}$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow \Theta : z}{\Gamma : e\ x \Downarrow \Theta : z} \quad \textit{Application}$$

$$\frac{\Gamma : e \Downarrow \Delta : z}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto z) : z} \quad \textit{Variable}$$

$$\frac{(\Gamma, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) : e \Downarrow \Delta : z}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e \Downarrow \Delta : z} \quad \textit{Let}$$

Figure 2.5: Launchbury's Operational Semantics

the implementation of languages such as GpH [103]. (See [36, 11] for more explanation on this.) Moreover, works such as [105, 106, 108] very well demonstrate the suitability of natural semantics for the source-level reasoning. This thesis also hopes to demonstrate this suitability.

[10] starts by offering a calculus for call-by-need. This is λ_{let} syntax and small-step operational step of which comes in Figure 2.6. Directly manipulating let-bindings as opposed to dealing with stores (like heaps in [55] or lists of bindings in [89, 95, 96]) is remarkable in λ_{let}. As the authors themselves explain, working without a separate store is one their aims; they consider this a lift in the level of operational semantics for λ-calculus. Ariola et al. show that λ_{let} enjoys confluence, as well as equivalence to call-by-need. [3] Their proofs rely on a heavy use of directed acyclic graphs enriched with boxes and labelled edges [9]. We believe this could have been greatly simplified by not dismissing stores. However, this suggests that their approach gives a more appropriate means for reasoning about languages semantics of which rely on graph term rewriting.

Generally, a deficiency of λ_{let} is its treatment for recursive values. In order to accommodate this, Ariola et al. suggest restricting substitutions to only occur when a variable appears in the hole of an evaluation context. The result is λ_ℓ — an operational semantics which looks like that of Launchbury [55]. λ_ℓ and Launchbury's operational semantics (Figure 2.5) are so similar that Ariola et al. also clarify it (Observation 8.2 in [10]):

Proposition. $\langle\Phi\rangle M \Downarrow \langle\Psi\rangle V$ iff $\lambda_\ell \vdash \mathsf{let}\ \Phi\ \mathsf{in}\ M \underset{need}{\longmapsto} \mathsf{let}\ \Psi\ \mathsf{in}\ V$.

(In words, $\langle\Phi\rangle M \Downarrow \langle\Psi\rangle V$ can be read as: evaluating M in the store Φ results in V with other effects reflected in Ψ.) They claim that the correctness proof of λ_ℓ is possible via showing its soundness and completeness with respect to Ariola and Klop's call-by-name with cycles [8].

[3] It is noteworthy that, as is traditional [85], their notion of equivalence only cares about reducibility. This is certainly weaker than the notions of equivalence that we use in this thesis. Furthermore, they only test reducibility for closed terms, whereas there is no such restriction in our notions. See Chapter 4.

Syntactic Domains

Variables	x, y, z		
Values	V, W	$::=$	$\lambda x.M$
Answers	A	$::=$	$V \mid \mathsf{let}\ x = M\ \mathsf{in}\ A$
Terms	L, M, N	$::=$	$x \mid V \mid M\ N \mid \mathsf{let}\ x = M\ \mathsf{in}\ N$

Axioms

(let-I)	$(\lambda x.M)\ N$	$= \mathsf{let}\ x = M\ \mathsf{in}\ N$
(let-V)	$\mathsf{let}\ x = V\ \mathsf{in}\ C[x]$	$= \mathsf{let}\ x = V\ \mathsf{in}\ C[V]$
(let-C)	$(\mathsf{let}\ x = L\ \mathsf{in}\ M)\ N$	$= \mathsf{let}\ x = L\ \mathsf{in}\ M\ N$
(let-A)	$\mathsf{let}\ y = (\mathsf{let}\ x = L\ \mathsf{in}\ M)\ \mathsf{in}\ N$	$= \mathsf{let}\ x = L\ \mathsf{in}\ \mathsf{let}\ y = M\ \mathsf{in}\ N$

Figure 2.6: λ_{let} of [10]

Syntax:

Expressions:	M	$::=$	$x \mid \lambda x.M \mid M\ M$
Value:	V	$::=$	$\lambda x.M$
Answers:	A	$::=$	$V \mid ((\lambda x.A)\ M)$
Evaluation Contexts:	E	$::=$	$[\,] \mid E\ M \mid (\lambda x.E[x])E \mid (\lambda x.E)M$

Axioms:

$(\lambda x.E[x])V$	$=$	$(\lambda x.E[V])V$	*deref*
$(\lambda x.A)M\ N$	$=$	$(\lambda x.A\ N)M$	*lift*
$(\lambda x.E[x])((\lambda y.A)M)$	$=$	$(\lambda y.(\lambda x.E[x])A)M$	*assoc*

Figure 2.7: λ_{need} of [7]

[7] is the next work of this group where Ariola and Felleisen present a slightly different operational semantics; See λ_{need} in Figure 2.7. They prove that their calculus is sound and complete with respect to Plotkin's call-by-name calculus [85]. In addition, Ariola and Felleisen show that their λ_{need} satisfies a Curry-Feys-style Standardisation Theorem. They also show that λ_{need} is a strict sub-theory of call-by-name. Some basic properties of λ_{need} are also established: confluence and Huet's notion of absence of critical pairs [44].

This paper offers a thorough explanation about their design decisions in which they compare environment-based implementations of laziness with graph-based ones. [7] has an emphasis on source syntax over graphs: they again employ enriched directed acyclic graphs which comes handy in their completeness proof. They show that this can be adapted for full-laziness as well [114, 39, 104, 46, 76]. Over the course of time, we now know that Launchbury's semantics (Figure 2.5) can also be neatly adapted for complete laziness [100].

It is noteworthy that, like their previous work [10], [7] has no proper treatment

Syntactic Domains

Variables	x, y, z		
Values	V, W	$::=$	$x \mid \lambda x.M$
Terms	L, M, N	$::=$	$V \mid M\ N \mid \mathsf{let}\ x = M\ \mathsf{in}\ N$

Reduction Rules

(I)	$(\lambda x.M)N$	$\to$	$\mathsf{let}\ x = N\ \mathsf{in}\ M$
(V)	$\mathsf{let}\ x = V\ \mathsf{in}\ C[x]$	$\to$	$\mathsf{let}\ x = V\ \mathsf{in}\ C[V]$
(C)	$(\mathsf{let}\ x = L\ \mathsf{in}\ M)\ N$	$\to$	$\mathsf{let}\ x = L\ \mathsf{in}\ M\ N$
(A)	$\mathsf{let}\ y = (\mathsf{let}\ x = L\ \mathsf{in}\ M)\ \mathsf{in}\ N$	$\to$	$\mathsf{let}\ x = L\ \mathsf{in}\ \mathsf{let}\ y = M\ \mathsf{in}\ N$
(G)	$\mathsf{let}\ x = M\ \mathsf{in}\ N$	$\to$	N **if** $x \notin \mathrm{fv}(N)$

Figure 2.8: λ_{NEED} of [62]

for recursive values. However, they report that they are in the process of finding one on the basis of [8, 9].

The third work of this group is [62] in which they present yet another variation of their original work [10]. See λ_{NEED} in Figure 2.8. λ_{NEED} enjoys pretty much every major interesting property of λ_{need} [7]. For example, Maraist et al. prove its confluence, and equivalence with Plotkin's call-by-name [85]. They also provide a computable deterministic strategy for standard evaluation. Whilst the proofs are mostly provided, some details are still left to [61]. Finally, it is noteworthy that this work also fails to provide a proper treatment for recursion.

Bastonero, Pravato, and Ronchi della Rocca (1997)

[15] aims to model lazy evaluation. However, like the work of Abramsky and Ong [2, 73, 3], it fails to capture sharing. Therefore, from the point of view of [7], Bastonero et al. provide another model for Plotkin's call-by-name λ-calculus which also corresponds to his respective evaluator [85]. Their model is based on an extension of coherent spaces [54] called pointed coherent spaces [42].

Bastonero et al. are interested in models which are adequate with respect to the operational semantics shown in Figure 2.1. Therefore, they acknowledge that the models offered by Absramsky and Ong [2, 73, 3] as well as the one by Longo [59] — referred to as $\mathcal{M}_{AO}$ and $\mathcal{M}_L$, respectively — do fulfill this. On the other hand, Bastonero et al. report that there are λ-theories which can be modelled in Scott's domains but not in coherent spaces [14], and vice-versa [13]. Thus, they focus on models based on coherent spaces.

With this in mind, they define the class of lazy regular models which enables them to present two models: $\mathcal{M}_1$ and $\mathcal{M}_2$. These are built over the coherent space which is the minimal solution to the equation $\mathcal{D} \approx \mathbb{I}\&(\mathcal{D} \Rightarrow_s \mathcal{D})$. (Here, $\mathbb{I}$ is the coherent space with just one atom, $\&$ is the additive Cartesian product, and $\mathcal{D} \Rightarrow_s \mathcal{D}$ is the coherent space representing the stable functions from $\mathcal{D}$ to $\mathcal{D}$.) Bastonero et al. prove that $\mathcal{M}_1$ has the same theory as $\mathcal{M}_L$ and conjecture that

$\mathcal{M}_2$ has the same theory as $\mathcal{M}_{AO}$. But both $\mathcal{M}_1$ and $\mathcal{M}_2$ have a local structure which is different from those of $\mathcal{M}_L$ and $\mathcal{M}_{AO}$. Interestingly enough, neither of $\mathcal{M}_1$ and $\mathcal{M}_2$ is fully abstract.

Moran and Sands (1999)

The operational semantics which [69] presents is based on Sestoft's "mark 1" abstract machine for laziness, which is essentially a Krivine-machine [17] with heap updating. Although Moran and Sands' small-step operational semantics (Figure 2.9) is inspired by that of Ariola et al. [10, 7, 62], their system has no problem with recursion. They also argue that the observational equivalence/approximation of Ariola et al. — which only observes termination — cannot distinguish between call-by-need and call-by-name. The key feature of their work, on the contrary, is its notion for a unit of cost: the abstract machine reduction step. It is based on this that Moran and Sands come up with their notion of improvement:

> M is improved by N if all program contexts $\mathbb{C}$, when $\mathbb{C}[M]$ terminates then $\mathbb{C}[N]$ terminates as cheaply.

This gives rise to a variety of technical results including an operational theory which retains the computational distinction between call-by-name and call-by-need.

Moran and Sands prove a context lemma for their semantics. Using their *tick algebra*, they show that their improvement relation has an inequational theory which validates the reduction rules of call-by-need calculi by Ariola et al [10, 7, 62]. They, next, prove a syntactic continuity property which characterises improvement of a recursive function in terms of its finite unwindings. [4] Finally, Moran and Sands present two proof techniques: they adapt the improvement theorem and improvement induction as formerly studied for call-by-name in [93] and [94], respectively. [5]

Some notation is necessary for Figure 2.9: each configuration is of the form $\langle \Gamma, M, S \rangle$. Γ is a heap of bindings, M is an expression, and S is a stack of variable names, case alternatives, or update markers — denoted by $\#x$ — for some variable x. $a : S$ is a pushed on top of S. $x \not\downarrow \mathrm{dom}(\Gamma, S)$ means x is neither in the domain of Γ, nor in the domain of S.

Ghaly et al. (2007)

[28] starts with a comparison between the operational semantics of Purushothaman and Seaman [89, 95, 96] and that of Launchbury [55]. They then offer yet another operational semantics in which λ-abstractions are not considered to be values. However, they still claim that their operational semantics describes lazy

[4] This forms the basis of fixed-point induction style proofs.

[5] These techniques verify the correctness and safety of recursion-based program transformations which proceed by local improvements. [69]

$$
\begin{array}{rcll}
\langle \Gamma\{x = M\}, x, S\rangle & \to & \langle \Gamma, M, \#x : S\rangle & (\textit{Lookup}) \\
\langle \Gamma, V, \#x : S\rangle & \to & \langle \Gamma\{x = V\}, V, S\rangle & (\textit{Update}) \\
\langle \Gamma, M\ x, S\rangle & \to & \langle \Gamma, M, x : S\rangle & (\textit{Unwind}) \\
\langle \Gamma, \lambda x.M, y : S\rangle & \to & \langle \Gamma, M[y/x], S\rangle & (\textit{Subst}) \\
\langle \Gamma, \mathsf{case}\ M\ \mathsf{of}\ \textit{alts}, S\rangle & \to & \langle \Gamma, M, \textit{alts} : S\rangle & (\textit{Case}) \\
\langle \Gamma, c_j\vec{y}, \{c_i\vec{x}_i \to N_i\} : S\rangle & \to & \langle \Gamma, N_j[\vec{y}/\vec{x}_j], S\rangle & (\textit{Branch}) \\
\langle \Gamma, \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N, S\rangle & \to & \langle \Gamma\{\vec{x} = \vec{M}\}, N, S\rangle \quad x \not\downarrow \mathrm{dom}(\Gamma, S) & (\textit{Letrec})
\end{array}
$$

Figure 2.9: The Abstract Machine of [69]

evaluation. They claim that their operational semantics is based on that of Purushothaman and Seaman with some minor differences. The relationship between their system and any other semantics for lazy evaluation is not clear though.

Discussion Although very powerful, the framework developed by Abramsky and Ong [2, 73, 3], as well as that of Bastonero et al. [15] do not form a good basis for our purpose because they do not model sharing. The work of Purushothaman and Seaman also do not form a good basis for our purpose either because they miss sharing for recursive functions. Moreover, the fact that they work on ordered bindings causes several difficulties which do not exist in Launchbury's work [55]. Ariola et al. [10, 7, 62] also fail to deal appropriately with recursion. On the other hand, Moran and Sand's interesting framework [69] for measuring the amount of sharing in call-by-need is more than what we need because we are not to provide similar measures for observational-equivalence in presence of selective-strictness. Finally, we do not choose to base our work on that of Ghaly at al. [28] because their work is not well-presented enough.

The only work which remains then is that of Launchbury [55]. It turn out that his framework is even suitable for extending this thesis to parallelism. Our evidence for this is that the operational semantics of parallel languages like GpH [103] are already based on Launchbury's semantics. Furthermore, in this thesis, we provide new means for examining the behaviour of programming languages which drastically depend on heaps. (See Chapters 7 onwards.) Launchbury's notion of heaps fits this application perfectly.

2.2 History of seq

Before considering the literature on selective strictness, we should give a brief account of the semantics of seq of Haskell.

As it turns out, the meaning of seq has undergone a change over time. Historically, seq was first given the following (asymmetric) semantics ([77, Section 6.2])

$$
\begin{array}{lll}
\mathsf{seq} & \bot & b = \bot \\
\mathsf{seq} & a & b = b \qquad \textit{if } a \neq \bot.
\end{array}
$$

From HASKELL 1.3, however, whilst the same term seq was preserved, it was employed with a slightly and subtly different semantics which is symmetric in its strictness specifications

$$\begin{aligned} \mathsf{seq}\ \bot\ \ a\ &= \bot \\ \mathsf{seq}\ a\ \ \bot\ &= \bot \\ \mathsf{seq}\ a\ \ b\ &= b. \end{aligned}$$

The old semantics of seq then tended to be given to pseq.

Interestingly, some HASKELL variations such as GPH or Eden were designed before this change in semantics. Therefore, their seq is not totally compatible with today's definition of seq in HASKELL. In other words, HASKELL is not backward compatible on seq. This backward incompatibility (since HASKELL 1.3) has caused a great confusion. It becomes especially confusing when one considers the research which predates HASKELL 1.3. For example [11] correctly target the seq of their time, but, rather address today's pseq.

This confusion is still present in the HASKELL community. One campaign [78, 63] believe that the denotational semantics of seq and pseq is the same whilst the only constraint on seq is that it is strict in both its argument (as chosen by HASKELL 1.3+). They argue that this is the reason why the operational semantics of seq is not portable across HASKELL implementations. This campaign believes in an implementation of pseq such as that of GHC [29]

$$\mathsf{pseq}\ x\ y = x\ \mathsf{seq}\ \mathsf{lazy}\ y$$

where $\mathsf{lazy}\ x = x$ but lazy has a special meaning to the strictness analyser in GHC: there is no demand inferred on x in $\mathsf{lazy}\ x$. The other campaign of the HASKELL community [64, 115] believe that pseq is to force evaluation of its first argument to be done before **starting** the evaluation of its second argument. Consequently, in a heap where evaluating x implies evaluation of y, trying x pseq y will result in a blackhole whilst x seq y might reduce. So, the subtle difference between seq and pseq in words turns out to be considerable in the operational behaviour. In fact, pseq demands a lock mechanism such as the $\overset{B}{\mapsto}$ bindings used in [11]. (See Figure 2.20 of this thesis for more.)

In this thesis, we model selective strictness for the original seq (that of HASKELL 1.3- which was asymmetric) or today's pseq. We make this decision because we had originally chosen to prove observational equivalence between GPH programs. Interestingly enough, the related work on selective strictness, as well as the work of Hall et al. [36, 11] all consider the same seq. We review the literature on selective strictness in Section 2.3.

2.3 Selective Strictness

In this section, we review the work of three different groups of scholars who have studied the impacts of selective strictness from different points of view. The reviews are ordered chronologically. We end our review over the work of each group with a discussion on the relation between the respective studies and that of ours.

```
e1 = seq ((\ (Just x) y -> x) Nothing) 3
e2 = seq ((\ (Just x) -> (\ y -> x)) Nothing) 3
e3 = (\ ~(x, Just y) -> x) (0, Nothing)
e4 = case 1 of
          x | x==z -> (case 1 of w | False -> 33)
                 where z = 1
          y -> 101
e5 = case 1 of
          x | x==z -> (case 1 of w | True -> 33)
                 where z = 2
          y -> 101
e6 = let  fac 0 = 1
          fac n = n * (fac (n-1))
     in fac 3
```

```
Semantics> mE e1 rho0          Hugs> e1
    3                              3
Semantics> mE e2 rho0          Hugs> e2
    Program error: {undefined}     Program error: {e2_v2550 Maybe_Nothing}
Semantics> mE e3 rho0          Hugs> e3
    Program error: {undefined}     Program error: {e3_v2558 (Num_fromInt instNum_v35 0,...)}
Semantics> mE e4 rho0          Hugs> e4
    Program error: {undefined}     Program error: {e4_v2562 (Num_fromInt instNum_v35 1)}
Semantics> mE e5 rho0          Hugs> e5
    101                            101
Semantics> mE e6 rho0          Hugs> e6
```

Figure 2.10: A worked out example of the unit tests provided in [37]

Harrison et al. (2002 and 2005)

Harrison, Sheard, and Hook [37] start with noting that strictness enforcing mechanisms in Haskell are: nested patterns, case expressions and `let` declarations, guards and `where` clauses on equations, strict constructor functions, the `newtype` datatype definition facility, and the `seq` operator. They give all these mechanisms a generic name: means for *fine control of demand* — all of which they study in [37]. To the best of our knowledge, this work is the first to pinpoint the necessity of special care in reasoning about explicit strictness in presence of mixed (strict/lazy) semantics. Although their work is based on a denotational semantics, they explain that the combinatorial interaction between all these strictness enforcing means makes a fully denotational approach non-trivial. Therefore, instead, they take what they call a *calculational* approach. That is, they explain every detail for how to implement a Haskell interpreter in Haskell itself; The interpreter is meant to implement the details of their denotational semantics. They do not prove any properties of their calculationally-specified denotational semantics. Nor do they prove correctness of their implementation, rather they report it to have undergone a considerable number of unit tests, some of which they also present in their paper.

Figure 2.10 shows the head-to-head comparison of the Haskell programs `e1` to `e6` between the famous Hugs interpreter [50] for Haskell and the computationally-specified denotational semantics of [37]. `rho0` is an environment and `mE` is the

function which performs evaluation of expressions in their system.

Harrison and Kieburtz [38] then choose to adhere to the P-logic [52] of the Programatica project[43]. They explain why P-logic can capture selective strictness and therefore be an appropriate modal logic for verifying the behaviour of lazy programs in presence of selective strictness. This time, they provide a self-contained description of a typed, denotational semantics for the following HASKELL fragment: abstraction, application and case expressions (without guards) [6]. Their denotational semantics is based on an extension to the *type frames* semantics of the simply-typed λ-calculus [34, 67] and is closely related to their previous interpreter [37]. They prove soundness of their P-logic inference rules with respect to this denotational semantics.

As a summary, we would like to remind that, as also explained in Section 2.1, the models based denotational semantics do not form a suitable basis for our purposes. This of course includes the work of Harrison et al. [37, 38] which are fully denotational.

van Eekelen and de Mol (2002, 2004, and 2007)

In the first work of their thread [105], van Eekelen and de Mol outline the development of the CLEAN programming language [84] which is coupled with the SPARKLE theorem prover [22]. The purpose of this mutual development is to state and prove properties of the program *on-the-fly*. They start by motivating the reasons why programmers use selective strictness, and the available selective strictness constructs. They define three sorts of strictness: mathematical, operational, and notational:

- A function f is *mathematically* strict in an argument x if the result of f applied to x is undefined if x is undefined.
- A function f is *operationally* strict in an argument x if the argument is always reduced to weak head normal form before the function application is evaluated and the result of the function is undefined if the argument is undefined.
- A function f is *notationally* strict in an argument x if the argument is somehow explicitly annotated as such.

Finally, using a variety of examples, they show how the use of some auxiliary functions can make formal reasoning about programs easier. They also show how their system can distinguish between "Ω" and "$\lambda.\Omega$" even within their language[7].

In [106] van Eekelen and de Mol start by noting reasons strictness is practiced and the means for this in HASKELL and CLEAN. It is this practice that motivates their study of the semantics of selective strictness. They first give a graph-based explanation on Launchbury's semantics [55]. Then they define strictness to be a property of a function. This does not seem to be the case for some other programming languages which support selective strictness such as GPH[103].

[6] This only includes the strictness enforcing machinery for pattern matching which, as Harrison and Kieburtz [38] also demonstrate, is in close interaction with datatypes.

[7] Here, Ω represents a denotationally $\perp$ computation.

$$\frac{(\Gamma, x_1 \mapsto e_1) : x_1 \Downarrow \Theta : z_1 \quad \Theta : e \Downarrow \Delta : z}{\Gamma : \mathsf{let!}\ x_1 = e_1\ \mathsf{in}\ e \Downarrow \Delta : z}\ \textbf{(StrictLet)}$$

Figure 2.11: van Eekelen and de Mol's Rule for let!

1. $\forall_{f,g}\forall_{xs}[map(f \circ g)xs = mapf(map\ g\ xs)]$
2. $\forall_{xs}[\text{FINITE}(xs) \Rightarrow reverse(reverse\ xs) = xs]$

Figure 2.12: Examples from [108] semantics of which changes by addition of strictness

The most significant point about [106] is that it correctly extends Launchbury's system for the case of explicit strictness. van Eekelen and de Mol's derivation rule for let! is depicted in Figure 2.11, where z represents a value. They also prove all the theorems stated in their previous work. Their most notable theorems here includes correctness and computational adequacy of their proposed operational semantics. Finally, they prove the following "folklore" pieces of knowledge:

- expressions that are bottom lazily, will also be bottom when we make something strict;
- when strictness is added to an expression that is non-bottom lazily, either the result stays the same or it becomes bottom;
- expressions that are non-bottom using strictness will also be non-bottom lazily with the same result.

The final work of van Eekelen and de Mol in this thread is [108] which is an updated version of the previous ones: It gives some short but helpful examples on when and why strictness annotation changes the semantics of program, and updates their literature review. Figure 2.12 shows two of such examples where FINITE(.) is a predicate which examines finiteness of its (list) argument.

As explained in Section 2.1, Launchbury's work offers the most complete platform for reasoning about the behaviour of lazy evaluation which happens to be the easiest for our purposes as well. The work of van Eekelen and de Mol correctly extend Launchbury's system to selective strictness. Therefore, although van Eekelen and de Mol never prove any identities using their system, we also choose to base our work on their system.

Johann, Voigtlander, and Seidel (2004, 2006, 2007, and 2009)

Johann and Voigtländer [47] focus on the effect of seq on free theorems. As first popularised by Wadler [113], free theorems are said to be *free* because they can be derived solely from the type of a function — virtually for free, i.e., with

Theorem 2.1. *For every function*

$$\mathit{filter} :: \forall\alpha.\ (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

and appropriately typed p, h, and l, the following hold:

- *If h is strict, then:*

$$\mathit{filter}\ p\ (\mathit{map}\ h\ l) \sqsubseteq \mathit{map}\ h\ (\mathit{filter}(p \circ h)\ l)$$

- *if $p \neq \bot$ and h is strict and total, then:*

$$\mathit{filter}\ p\ (\mathit{map}\ h\ l) \sqsupseteq \mathit{map}\ h\ (\mathit{filter}\ (p \circ h)\ l)$$

- *if $p \neq \bot$ and h is strict and total, then:*

$$\mathit{filter}\ p\ (\mathit{map}\ h\ l) = \mathit{map}\ h\ (\mathit{filter}\ (p \circ h)\ l).$$

Figure 2.13: A Free Theorem that Holds in Presence of seq.

no knowledge of the actual definition of the function. Free theorems are used to derive program equivalences involving parametric polymorphic functions in programming languages based on Girard-Reynolds [31, 91] Lambda Calculus. Johann and Voigtländer show that although free theorems hold unconditionally for polymorphic functions in Girard-Reynolds calculus, they might fail in presence of strict primitives such as seq. They explain why seq causes this failure, and, develop a technique for recovering the free theorems failing in such situations. They also apply their technique to prove some famous free theorems as shown in Figure 2.13. As they explain, this technique can be used for recovering free theorems in a language as large as a subset of Haskell [77] that corresponds to Girard-Reynolds-style calculus with fixpoints, plus algebraic data types, and seq.

In their second related work [48], Johann and Voigtländer focus on an important application of free theorems, namely, short cut fusion [8] [30] and more generally the related issues to program transformation. They use the results developed in their previous work [47] to study the problematic aspects caused by seq for the free theorems associated with the *foldr/build* rule used in short cut fusion, as well as its dual — the *destroy/unfoldr* rule [101]. In both cases they demonstrate the usefulness of their machinery for recovering from the side effects. See Figure 2.14 for example. [9] The results of this latter paper are valid for a subset of Haskell as large as Pitts' PolyFix [80] with seq. It is also noteworthy

[8] Short cut fusion is an optimisation technique, gist of which being the use of some local replacement rules to avoid unnecessary construction/destruction of intermediate lists.

[9] Interestingly enough, *build* in Figure 2.14 has a rank-2 type [57], i.e., it takes a polymorphic function as an argument. Haskell's current type-system does not support this. However, most Haskell implementations do support it as an extension. See [112] for more.

Theorem 2.2. *For every closed type τ, every function*

$$g :: \forall\beta.\ (\tau \to \beta \to \beta) \to \beta \to \beta,$$

and appropriately typed c and n, the following hold:

- $\mathit{foldr}\ c\ n\ (\mathit{build}\ g) \sqsupseteq g\ c\ n$
- *if* $c\ \bot\ \bot \neq \bot$ *and* $n \neq \bot$, *then*

$$\mathit{foldr}\ c\ n\ (\mathit{build}\ g) = g\ c\ n.$$

Figure 2.14: Recovered Free Theorem for *foldr* in [48]

Let $\tau \in \mathit{Typ}$ and $R, R' \in \mathit{Term}(\tau)$. Write $R \rightsquigarrow R'$ for the following pairs:

R	R'	if
$(\lambda x :: \tau'.N)\ A$	$N[A/x]$	$x :: \tau' \vdash N :: \tau$
$(\Lambda\alpha.N)_{\tau'}$	$N[\tau'/\alpha]$	$\alpha \vdash N :: \tau''$
$\mathsf{case}\ \mathsf{nil}_{\tau'}\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow M; h : t \Rightarrow M'\}$	M	$h :: \tau', t :: \tau'\text{-}\mathit{list} \vdash M' :: \tau$
$\mathsf{case}\ H : T\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow M; h : t \Rightarrow M'\}$	$M'[H/h, T/t]$	$h :: \tau', t :: \tau'\text{-}\mathit{list} \vdash M' :: \tau$
$\mathsf{fix}(F)$	$F\ \mathsf{fix}(F)$	
$\mathsf{seq}\ (V, M)$	M	$V \in \mathit{Value}$,

where x, h, and t are term variables, α is a type variable, $\tau' \in \mathit{Typ}$, $A, H \in \mathit{Term}(\tau')$, $M \in \mathit{Term}(\tau)$, $T \in \mathit{Term}(\tau'\text{-}\mathit{list})$, $F \in \mathit{Term}(\tau \to \tau)$, and further types and terms that occur in the table are subject to the restrictions recorded on the right.

Figure 2.15: Operational Semantics of PolySeq

that [48] covers all the proofs omitted in [47].

In their next work [110], Voigtländer and Johann explain that a denotational model would depend on correctness of some open conjectures. Therefore, they choose to work in an operational setting. They offer PolySeq for this purpose and for reasoning *equationally*. Voigtländer and Johann report that the shift to reasoning equationally is only a part of a longer plan for adapting operational settings to analysing free theorems in presence of selective strictness.

PolySeq's syntax is that of Pitts' PolyPCF [80] plus seq. However, unlike PolyPCF, PolySeq has a small-step operational semantics which comes in Figure 2.15. As Voigtländer and Johann explain, as long as free theorems are concerned, there is no difference between call-by-need and call-by-name. Therefore, to avoid complications, they design PolySeq so that it does not capture sharing. It is not a surprise then that they have a result like this [110, Corollary 7.5]:
Proposition. For every $A, B \in \mathit{Term}$, if $A \Downarrow$, then $\mathsf{seq}\ (A, B) =_{obs} B$.

[110] shows how to prove the correctness, with respect to observational equivalence, of parametricity-based transformations on PolySeq programs. Voigtaländer and Johann's example for this is again short-cut fusion. As they explain, their results work for implementations of Haskell like GHC [29]. The special point about GHC is that it combines Haskell's selective strictness with impredicative polymorphism [112]. [10]

The forth related work of Voigtaländer and Johann [111] goes back to considering observational approximation but still in an operational setting. [111] is mainly about constructing a parametric model of observational approximation for PolySeq. They show the correctness of their technique, and, in order to demonstrate it, they again examine the short cut fusion. [111] omits the proofs which are already done in [110].

The last work of this thread is [97] where Seidel and Voigtländer combine the ideas of [47] and [56] to offer a new type-system. Their new type-system helps them to remove some unnecessary side-conditions for free theorems in presence of selective strictness. The solution provided in [97] is two-fold: they first refine PolySeq's type-system to come up with PolySeq*. Seidel and Voigtländer prove that PolySeq* and PolySeq are equivalently expressive. However, PolySeq* provides a stronger parametricity theorem with a better algorithmic behaviour. Next, they provide PolySeq^C which they show to be equivalently expressive to PolySeq* but has yet more refined type-system. The special use of PolySeq^C is that it enables deriving minimal — in the sense of minimal logical relations — free theorems about the programs equally typable in PolySeq. This is through solving and minimalising the set of additional constraints that PolySeq^C generates during the typing of expressions. Some long proofs of [97] are presented in [98].

We note that Johann, Voigtländer, and Seidel's study of free theorems in presence of selective strictness is broader than our study in this thesis. Additionally, their research addresses more sizeable languages than ours. However, their machinery targets inequational theorems, which technically means that they do observational approximation rather than observational equivalence. Moreover, their approach substantially relies on certain preconditions to hold so that they can recover free theorems. It turns out that, to recover equational versions, these preconditions need to be even stronger.

As an example, Figure 2.16 depicts the free theorem derivable for `seq` in PolySeq, as provided by Johann and Voigtländer's online demo[11]. (Note that, unlike us that use `seq` in an infix form, Johann and Voigtländer use it in a prefix form.) Taking $f(x) \equiv x$ and $g(x) \equiv x$ `seq` z, if it would have been the case that g was total, associativity of `seq` would have followed in PolySeq as well. However, g is obviously not total. [12] With the aids of the techniques developed in [97], Seidel and Voigtländer remove the need for g to be total. Yet, as provided

[10] In short, predicative polymorphism means that one can only instantiate polymorphic functions with monomorphic types. Impredicative polymorphism is removing this restriction in instantiation of polymorphic functions.

[11] http://linux.tcs.inf.tu-dresden.de/~voigt/ft/

[12] Call a relation $\mathcal{R}$ **total** when for every pair x, y such that $x \mathcal{R} y$, $x \neq \bot$ implies $y \neq \bot$.

The theorem generated for functions of the type

$$seq :: \forall ab.a \to b \to b$$

in the sublanguage of Haskell with general recursion and selective strictness, equational style, is:

$$\begin{array}{l}\forall t_1, t_2 \in \text{TYPES}, R \in \text{REL}(t_1, t_2), R \textit{ strict, continuous, and bottom-reflecting.} \\ \quad \forall t_3, t_4 \in \text{TYPES}, S \in \text{REL}(t_3, t_4), S \textit{strict, continuous, and bottom-reflecting.} \\ \quad\quad \forall (x, y) \in R. \\ \quad\quad\quad \forall (z, v) \in S.(\mathsf{seq}\ \ x\ z, \mathsf{seq}\ \ y\ v) \in S\end{array}$$

Reducing all permissible relation variables to functions yields:

$$\begin{array}{l}\forall t_1, t_2 \in \text{TYPES}, f :: t_1 \to t_2, f \textit{ strict and total.} \\ \quad \forall t_3, t_4 \in \text{TYPES}, g :: t_3 \to t_4, g \textit{ strict and total.} \\ \quad\quad \forall x :: t_1.\ \forall y :: t_3.\ g\ (\mathsf{seq}\ \ x\ y) = \mathsf{seq}\ \ (f\ x)\ (g\ y)\end{array}$$

Figure 2.16: A Free Theorem about seq

by their online demo[13], for $f :: t_1 \to t_2$ and $g :: t_3 \to t_4$ they still demand $\mathsf{seq}_{t_1,t_3} \neq \bot \Leftrightarrow \mathsf{seq}_{t_2,t_4} \neq \bot$ so that they can derive associativity of seq. [14] Similar constraints on types seem natural for free theorems because they are essentially derived from the types. On the contrary, we put no constraints on the types and our system works independent of typing.

As a final comment, we would like to remind that the majority of works by this group take denotational approaches which despite their undeniable merit, as explained in Section 2.1, do not form a good basis for the properties we are after. On the other hand, the only two works of theirs which take an operational approach ([110] and [111]) are also not suitable for our purposes because they are designed not to capture sharing. Technically, this means that they study selective strictness for call-by-name as opposed to call-by-need.

2.4 Other Related Work

This section reviews further sets of related work in a chronological order. In outline, [116] provides an alternative mechanism for studying sharing; Hall et al. [36, 11] extend Launchbury's semantics to generalise the study of selective strictness to that of parallel coordination; Hidalgo-Herrero [40] offers a denotational semantics in which they can prove similar identities to the ones we prove in this thesis; [18] studies reasoning about partial languages in the presence of selective strictness — although selective strictness by itself is not a concern of their research; Formal reasoning about functional programming is investigated in

[13] http://linux.tcs.inf.tu-dresden.de/~seideld/cgi-bin/polyseq.cgi

[14] In this notation, f_t denotes a polymorphic function f instantiated for type t.

[20]; Finally, [92] extends Launchbury's operational semantics [55] for the study of distributed lazy evaluation.

Yoshida (1993)

[116] presents a weak λ-calculus that formalises functional execution with shared environments. This is called λf which Yoshida proves to be tractable. She employs explicit substitutions which is reminiscent of the common practice of using let-binding in functional programming. (λf is a calculus in that it comes with no reduction strategy.) She established the Church-Rosser property and normalisability of λf. Based on those, Yoshida then shows how the left-most reduction strategy of her calculus is optimal in the weak execution scheme. Whilst proof sketches come in [116], the complete proofs come in [117].

It is a fact that, for an arbitrary expression, the first occasion in which a variable is evaluated is not known until the run-time. Therefore, as also explained in [95, 96, 62], modelling lazy evaluation is not possible via simply providing a reduction strategy for the work like [1] with semantics based on explicit substitution. This is because such works simply copy the contents of substitutions for the case of application:

$$App \quad (a\ b)[s] \to (a[s])(b[s]).$$

Yoshida is careful to design a calculus which works perfectly for its purpose, i.e., optimal reduction. For this purpose, in certain situations, variables are fine not to be updated with the respective substitutions. However, this does not completely fit lazy evaluation where each variable is needed to be evaluated at most once. Besides, due to her notion of observation, reduction in λf is not equivalent to reduction to weak-head normal form (whnf) as is used in lazy programming languages.

Hall et al. (1998 and 2000)

[36] is the first work in this thread, which starts by noting that parallel programs, in addition to computation, must also specify *coordination*. That is, more than just *what* to compute (which is of concern to any program), parallel programs are concerned about *how* to arrange the computation too. The authors use the two primitives seq and par of GpH[58] as the means for this purpose, with definitions:

$$\begin{aligned} e_0 \ \mathsf{par}\ e_1 &= e_1 \\ e_0 \ \mathsf{seq}\ e_1 &= \begin{cases} \bot & \text{if } e_0 = \bot \\ e_1 & \text{otherwise} \end{cases} \end{aligned}$$

(As discussed in Section 2.2, this asymmetric definition for seq agrees with the respective definition of Haskell standard of the time [36] was written.)

However, they express their wish in their model to be general enough for investigation on the validity of identities such as the ones in Figure 2.22. [15] For

[15] Hall et al. however do not suggest using the $[\![.]\!]$ denotation for establishing the identities.

$$
\begin{array}{rcl}
w, x, y, z \in \mathsf{Var} & & \\
e \in \mathsf{Exp} & ::= & \lambda x.e \\
& | & y\ x \\
& | & x \\
& | & \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e \\
& | & x\ \mathsf{par}\ e \\
& | & x\ \mathsf{seq}\ e \\
\Delta, \Theta \in \mathsf{Heap} & = & \mathsf{Var} \rightharpoonup \mathsf{Exp} \\
v \in \mathsf{Val} & ::= & \lambda x.e
\end{array}
$$

Figure 2.17: The Syntax of Hall et al. [36]

that, their model is of a slightly higher level of abstraction than the mere "programming language" GpH. That is, as high as speculative evaluation only. (See our review on their next work [11] which is closer to the actual GpH programming language.) Their semantics is an extension over that of Launchbury [55]. One key difference between theirs and the latter is that their semantics is a small-step one. They argue that, because of the coordination requirements between thread creation and destruction, consideration of behaviour at the level of single reduction is needed. Therefore, their choice for small-step semantics fits naturally to the needs of parallelism.

Figures 2.17 and 2.18 show their syntax and operational semantics, respectively. [16] For explanations on their rules, consult [36] where they also offer sample reductions. The similarities between their work and that of Launchbury does not stop here in that they prove: Firstly, that in the case of single resource (when $|\Gamma| = 1$ in rule (**Product**) of Figure 2.18) their system simulates the latter one. And, secondly, that in the presence of more than one resource, the computations are fully serialisable. Consequently, even in the presence of parallelism, their systems correctly models sharing.

The next related work of this group is [11]. This work is much closer to the real operational semantics of GpH[58] in that, for instance, the bindings are labeled to model GpH threads — as (A)ctive, (R)unnable, (I)nactive, and (B)locked — and, the semantics is parameterised with the number of processors (N). Like [36], they present their semantics in a two-level transition system; the lower level pursues single-thread transitions, where the upper level manages their combination. Although their semantics is still small-step, it is at a higher level than most abstract machines. For example, it has no need for stacks or blocking queues.

Figure 2.19 shows the syntax of this work, and Figure 2.20 depicts part of their operational semantics. Note the following notational conventions in these Figures: (1) $x \overset{\alpha}{\mapsto} e$ stands for $x \mapsto (e, \alpha)$, (2) $e^x ::= x \mid x\ y \mid x\ \mathsf{seq}\ e'$. The most related part of Figure 2.20 is its (*seq*) and (*seq-elim*) rules. There are two differences between this couple of rules and our (**seq**) rule. (See Definition 3.4 for

[16] Note that despite Launchbury who employs z for values (whnf expressions), they employ v for this purpose.

Sequential Rules

$$\Delta : (y \mapsto \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e) \rightarrow (\Delta, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) : (y \mapsto e) \quad \textbf{(Let)}$$

$$(\Delta, x \mapsto \lambda w.e) : (z \mapsto x\ y) \rightarrow (\Delta, x \mapsto \lambda w.e) : (z \mapsto e[y/w]) \quad \textbf{(Application)}$$

$$(\Delta, x \mapsto v) : (y \mapsto x) \rightarrow (\Delta, x \mapsto v) : (y \mapsto \hat{v}) \quad \textbf{(Variable)}$$

$$(\Delta : x \mapsto v) : (z \mapsto x\ \mathsf{seq}\ e) \rightarrow (\Delta, x \mapsto v) : (z \mapsto e) \quad \textbf{(Sequence)}$$

$$\Delta : (z \mapsto x\ \mathsf{par}\ e') \rightarrow \Delta : (z \mapsto e') \quad \textbf{(Parallel1)}$$

$$(\Delta, x \mapsto e) : (z \mapsto x\ \mathsf{par}\ e') \rightarrow \Delta : (x \mapsto e, z \mapsto e') \quad \textbf{(Parallel2)}$$

Parallel Rules

$$\frac{\Delta : \tau_i \rightarrow \Delta : \tau_i', \quad 1 \leq i \leq n_{red} \quad n_{red} + m_a \leq \mathbf{max}}{\Delta : \Gamma \rightarrow \Gamma_d \cup \bigcup \Delta_i \setminus \Delta_a : \Delta_a \cup \bigcup \tau_i'} \quad \textbf{(Product)}$$

Figure 2.18: The Operation Semantics of Hall et al. [36]

$$\begin{array}{rl}
x, y, z & \in \mathsf{Var} \\
n & \in \mathsf{Number} \\
e & \in \mathsf{Exp} \\
e & ::= n \mid x \mid e\ x \mid \lambda x.e \mid \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e \\
 & \quad \mid e_1\ \mathsf{seq}\ e_2 \mid x\ \mathsf{par}\ e \\
H, K & \in \mathsf{Heap} = \mathsf{Var} \rightharpoonup (\mathsf{Exp}, \mathsf{State}) \\
\alpha, \beta & \in \mathsf{State} \\
\alpha & ::= \mathit{Inactive} \mid \mathit{Runnable} \mid \mathit{Active} \mid \mathit{Blocked} \\
v & ::= n \mid \lambda x.e
\end{array}$$

Figure 2.19: The Syntax of [11]

$$H : z \overset{A}{\mapsto} \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e \longrightarrow (\{x_i \overset{I}{\mapsto} e_i\}_{i=1}^{n}, z \overset{A}{\mapsto} e) \qquad (let)$$

$$(H, x \overset{I}{\mapsto} v) : z \overset{A}{\mapsto} x \longrightarrow (z \overset{A}{\mapsto} \hat{v}) \qquad (var)$$

$$(H, x \overset{I}{\mapsto} e) : z \overset{A}{\mapsto} x \longrightarrow (x \overset{R}{\mapsto} e, z \overset{B}{\mapsto} x) \qquad (block_1)$$

$$(H, x \overset{RAB}{\mapsto} e) : z \overset{A}{\mapsto} x \longrightarrow (z \overset{B}{\mapsto} x) \qquad (block_2)$$

$$H : z \overset{A}{\mapsto} (\lambda y.e)\ x \longrightarrow (z \overset{A}{\mapsto} e[x/y]) \qquad (subst)$$

$$\frac{H : z \overset{A}{\mapsto} e \longrightarrow (K, z \overset{\alpha}{\mapsto} e')}{H : z \overset{A}{\mapsto} e\ x \longrightarrow (K, z \overset{\alpha}{\mapsto} e'\ x)} \qquad (app)$$

$$H : z \overset{A}{\mapsto} v\ \mathsf{seq}\ e \longrightarrow (z \overset{A}{\mapsto} e) \qquad (seq\text{-}elim)$$

$$\frac{H : z \overset{A}{\mapsto} e_1 \longrightarrow (K, z \overset{\alpha}{\mapsto} e_1')}{H : z \overset{A}{\mapsto} e_1\ \mathsf{seq}\ e_2 \longrightarrow (K, z \overset{\alpha}{\mapsto} e_1'\ \mathsf{seq}\ e_2)} \qquad (seq)$$

$$(H, x \overset{RAB}{\mapsto} e_1) : z \overset{A}{\mapsto} x\ \mathsf{par}\ e_2 \longrightarrow (z \overset{A}{\mapsto} e_2) \qquad (par\text{-}elim)$$

Figure 2.20: Part of the Semantics of [11]

our (**seq**) rule.) Firstly, this is a small-step treatment of Haskell's seq operator whereas we deal with the same operator in a big-step way. Secondly, for the sake of modelling parallelism as well as selective strictness, their bindings are labeled (with $\overset{A}{\mapsto}$ and $\overset{\alpha}{\mapsto}$ in this case).

In this work of theirs, they extend Launchbury's denotational semantics for the case of par and seq. With respect to this, they prove Soundness, Computational Adequacy, and Determinacy of their system. Most interesting out of that list, is perhaps Determinacy which states that "the same result is always obtained irrespective of the number of processors, and irrespective of which runnable threads are chosen for activation during the computation". Afterwards, they show the richness of their semantics in that, not only it can model GpH, but — with vary small manipulations — it can also reflect the behaviour of other evaluation strategies such as Sequential Evaluation, Fully Speculative Evaluation [33], Non-deterministic Choice [65, 68], and Controlled Speculative Evaluation. They finalise this work by formally defining Work, Runtime, and Average Parallelism (all of which they had only informally defined in their previous work [36]), in addition to Maximum Parallelism.

In summary, the work of Hall et al. [36, 11] suggest an interesting framework for extending our study of observational equivalence to include coordination as well as selective strictness. The second and third identities in Figure 2.22 are examples of observational equivalence in such a framework where par is a coordination mechanism. This is subject to future work.

Semantic Domains

$$
\begin{array}{rcll}
 & \mathbf{Cont} & = \mathbf{Env} \to \mathbf{Env} & \text{continuations} \\
\kappa & \in \mathbf{ECont} & = \mathbf{EVal} \to \mathbf{Cont} & \text{expression continuations} \\
\rho & \in \mathbf{Env} & = \mathbf{Ide} \to (\mathbf{Val} + \{\text{undefined}\}) & \text{environments} \\
v & \in \mathbf{Val} & = \mathbf{Eval} + \mathbf{Clo} + \{\text{not\_ready}\} & \text{values} \\
\epsilon & \in \mathbf{Eval} & = \mathbf{Abs} & \text{expressed values} \\
\alpha & \in \mathbf{Abs} & = \mathbf{Ide} \to \mathbf{Clo} & \text{abstract values} \\
\nu & \in \mathbf{Clo} & = \mathbf{ECont} \to \mathbf{Cont} & \text{closures}
\end{array}
$$

Evaluation Function

$$
\begin{array}{l}
\mathcal{E}[\![x]\!]\kappa = \mathsf{force}\ x\ \kappa \\
\mathcal{E}[\![\lambda x.e]\!]\kappa = \kappa(\lambda x.\mathcal{E}[\![e]\!]) \\
\mathcal{E}[\![x_1\ x_2]\!]\kappa = \mathcal{E}[\![x_1]\!]\kappa' \\
\quad \text{where } \kappa' = \lambda\epsilon.\lambda\rho.\epsilon\ x_2\ \kappa\ \rho
\end{array}
\qquad
\begin{array}{l}
\mathcal{E}[\![x_1\ \mathsf{seq}\ x_2]\!]\kappa = \mathcal{E}[\![x_1]\!]\kappa' \\
\quad \text{where } \kappa' = \lambda\epsilon.\lambda\rho.\mathcal{E}[\![x_2]\!]\ \kappa\ \rho \\
\mathcal{E}[\![x_1\ \mathsf{par}\ x_2]\!]\kappa = \mathcal{E}[\![x_2]\!]\kappa' \\
\quad \text{where } \left\{ \begin{array}{l} \kappa' = \lambda\epsilon.\lambda\rho.\kappa\ \epsilon\ \rho_{par} \\ \rho_{par} = \mathsf{par}\ x_1\ \rho \end{array} \right.
\end{array}
$$

$$
\mathcal{E}[\![\mathsf{let}\ \{x_i = e_i\}_{i=1}^n\ \mathsf{in}\ x]\!]\kappa = \lambda\rho.\mathcal{E}[\![x]\!]\kappa\ \rho'
\quad \text{where } \left\{ \begin{array}{l} \{y_i\}_{i=1}^n = \mathsf{newIde}\ n\ \rho \\ \rho' = \rho \oplus \{y_i \mapsto \mathcal{E}[\![e_i[y_j/x_j]_{j=1}^n]\!] \mid 1 \le i \le n\} \end{array} \right.
$$

Figure 2.21: Denotational Semantics of Hidalgo-Herrero [40]

Hidalgo-Herrero (2004)

Using the fully denotational approach shown in Figure 2.21, Hidalgo-Herrero [40] proves a number of denotational identities about programs containing seq and a parallel composition, par. The key identities are shown in Figure 2.22 where $\mathcal{E} :: \mathbf{Exp} \to \mathbf{ECont} \to \mathbf{Cont}$. The way ρ_0 is defined turns out to be more subtle than it might be expected. Although the model of Hidalgo-Herrero [40] does not capture sharing, it still considers more than our $\cong_v$ (Definition 4.1), i.e., simply the final value of an expression. However, unlike our $\cong_s$ (Definition 4.3), they do not observe the entire environment (or heap in Launchbury's semantics [55]). For example, for the first identity of Figure 2.22, x, y, and z are observed whilst other implicated changes to the environment are not considered. Therefore, currently, it is unclear what the correspondence is between their denotational identity and the operational equivalences in this thesis. This is future work.

$$
\begin{array}{l}
\mathcal{E}[\![x\ \mathsf{seq}\ (y\ \mathsf{seq}\ z)]\!]\ \kappa_0\ \rho_0 = \mathcal{E}[\![(x\ \mathsf{seq}\ y)\ \mathsf{seq}\ z]\!]\ \kappa_0\ \rho_0 \\
\mathcal{E}[\![x\ \mathsf{par}\ (x\ \mathsf{par}\ y)]\!]\ \kappa_0\ \rho_0 = \mathcal{E}[\![x\ \mathsf{par}\ y]\!]\ \kappa_0\ \rho_0 \\
\mathcal{E}[\![x\ \mathsf{seq}\ (y\ \mathsf{par}\ z)]\!]\ \kappa_0\ \rho_0 = \mathcal{E}[\![(x\ \mathsf{seq}\ y)\ \mathsf{par}\ (x\ \mathsf{seq}\ z)]\!]\ \kappa_0\ \rho_0
\end{array}
$$

For appropriate κ_0 and ρ_0.

Figure 2.22: Related Identities Proved in [40]

Danielsson, Hughes, Jansson, and Gibbons (2006)

[18] aims to justify the common practice in assuming that equational laws in total languages are also correct in partial ones. For this purpose, they define two languages with the same syntax, but one total and the other partial. The key result is then to show that when two closed terms have the same semantics in the total language, they will have *related* semantics in the partial one.

This is achieved through defining two denotational semantics for their languages: a set-theoretic ($[\![.]\!]$) and a model-theoretic ($(\!|.|\!)$). They then define their moral equality relation ($\sim$):

- If a type σ does not contain function spaces, then $x \sim_\sigma y$ iff x and y are equal total values.
- For functions, $f \sim g$ iff f and g map (total) related values to related values.

Using this, finally, they show that for any two terms t_1 and t_2, two pairs of contexts ρ_1, ρ_1' and ρ_2, ρ_2' both satisfying a certain condition, $(\!|t_1|\!)\rho_1' = (\!|t_2|\!)\rho_2'$ implies $[\![t_1]\!]\rho_1 \sim [\![t_2]\!]\rho_2$. In words, they prove that if two terms are equal in the set-theoretic world, then they are morally equal in the model-theoretic world. As an example of this, they demonstrate that

$$(\!|revMap \circ mapRev|\!)[y \mapsto n] = id \Rightarrow [\![(revMap \circ mapRev)\ xs]\!][y \mapsto n] = [\![xs]\!]$$

for $revMap = reverse \circ map(\lambda x.x - y)$, $mapRev = map(\lambda x.x + y) \circ reverse$, and some natural number n. The commonality between [18] and this thesis is that selective strictness is available in both languages studied in the former. However, there is no special focus on the role of selective strictness in their work. Despite that, the existence of `seq` in their syntax is perhaps because `seq` is one possible mechanism to produce partial definitions in Haskell.

de Mol (2009)

de Mol's PhD thesis [20] is on formal reasoning about functional programming. More specifically, it is on a tool dedicated to this purpose: the Sparkle proof assistant, mutually developed with Clean. The reason why [20] remains related to this thesis is that complete Sparkle proofs underpin how a program deals with $\bot$ as well as the circumstances under which a program yields $\bot$. In other words, they have special means for dealing with definedness properties of programs.

de Mol's thesis starts by giving a history of software bugs and the methods to cure/prevent them. Amongst those methods, de Mol defines formal reasoning to be "the mathematical process of building proofs", and explains why they chose this option. The remainder of his thesis comprises of nine papers de Mol has either been a major author of or has co-authored. Three of these are already studied in Section 2.3 because they were dedicated to selective strictness [105, 106, 108]. Here, we review the rest, each of which comes as a chapter in [20].

The first paper [21] reports the development of CleanProverSystem — a prototype for Sparkle — which works for a small subset of Clean. Example

$$\forall\alpha\forall xs \in \alpha\text{-}list.\ reverse\ (reverse\ xs) = xs \tag{2.1}$$
$$\forall\alpha\forall xs \in \alpha\text{-}list\forall n \in \mathbb{N}.\ (take\ n\ xs) +\!\!+ (drop\ n\ xs) = xs \tag{2.2}$$
$$\forall x, y, z \in \mathbb{N}.\ x^{(y+z)} = x^y \times x^z \tag{2.3}$$
$$\forall x \in \mathbb{N}.\ log\ 2^x = x \tag{2.4}$$

Figure 2.23: Some Theorems Proved in CleanProverSystem

theorems proved in CleanProverSystem are shown in Figure 2.23 but side conditions like $n \neq \perp$ are not considered for instance for (2.2). A more complete list of theorems proved in CleanProverSystem can be found in [19] which contains all the 72 theorems in Bird's famous textbook [16]. CleanProverSystem is still available online.[17]

The second paper [22] explains why the integration of Sparkle with Clean is preferred over indirection via common theorem provers such as Pvs [74], Coq [102], and Isabelle [75]. This is because Sparkle itself has a semantic based on lazy graph term-rewriting which allows reasoning to take place directly on Clean programs rather than their translations. Yet, Sparkle reasons on the translations of Clean programs into Core-Clean. Core-Clean is a subset of Clean which only supports application, sharing, and case distinction with a special support for strictness annotation. 80% of theorems proved in Bird's textbook [16] are reported in [22] to be successfully proved using Sparkle. Most importantly, the side condition $n \neq \perp$ is now added in for (2.1).

The third paper [24] is a tutorial of Sparkle for (functional) programmers. It first explains the concept of formal reasoning in general. Several design decisions are particularly justifies for Sparkle then — most importantly the integration of Sparkle in the Clean development studio. Finally, a brief tutorial on how to use Sparkle in the basic and advanced level is given. Demonstrating the effects of laziness and strictness annotation on Sparkle is of special interest to this tutorial.

de Mol starts his next chapter [107] by exemplifying the reformulation needed in the statement of theorems such as (2.1) upon changes in strictness properties. Next, it shows how Sparkle's support for explicit strictness facilitates such reformulations using appropriate modifications in proof rules. Two expressions are considered interchangeable in [107] when they either both compute the exact same value, or they do not reduce at all. Interestingly enough, our $\approx_v$ (Definition 4.1) demands the same condition to hold between two expressions but for evaluation in every heap.

In [25], a confluent term graph rewriting system is presented which is similar to Launchbury's operational semantics for lazy evaluation (Fig. 2.5). The rewrite system of [25], however, has a small-step semantics and therefore allows partial and inner reducts as well. It is also explained in [25] why their freedom of choice of redexes forms a suitable basis for formal reasoning. Finally, they show that a

[17] http://www.cs.ru.nl/~maartenm/CleanProverSystem/

Definitions:

$$O_1 \sim O_2 \Leftrightarrow (\forall o_1 \in O_1 \exists o_2 \in O_2.[o_1 \sqsubseteq o_2]) \wedge (\forall o_2 \in O_2 \exists o_1 \in O_1.[o_2 \sqsubseteq o_1])$$
$$\psi_1 \cong \psi_2 \Leftrightarrow Output(\psi_1) \sim Output(\psi_2)$$
$$Equal(e_1, e_2) \Leftrightarrow \forall x \in \mathcal{V}_e \forall e \in \mathcal{E}^{\mathsf{opened}}$$
$$[Output(\mathsf{run}\ e_{x \mapsto e_1}\ \mathsf{with}\ \psi) \sim Output(\mathsf{run}\ e_{x \mapsto e_2}\ \mathsf{with}\ \psi)]$$

The following terms are *semantically equal*

$$repeat\ 1 \tag{2.5}$$
$$\mathsf{let}\ x = [1,1]{::}x\ \mathsf{in}\ [1]{::}x \tag{2.6}$$
$$\mathsf{let}\ x = [1,1,1,1]{::}x\ \mathsf{in}\ x \tag{2.7}$$
$$repeat\ 1\ +\!\!+\ [2,3,4] \tag{2.8}$$
$$repeat\ 1\ +\!\!+(\mathsf{let}\ x = x\ \mathsf{in}\ x) \tag{2.9}$$

Figure 2.24: Semantical Equality in Chapter 8 of de Mol's Thesis

special strategy for their system behaves correctly with respect to Launchbury's operational semantics.

The contents of the next chapter in de Mol's thesis [20] is a simplified version of Chapter 7 in [23] which focuses on a formalised semantics for expression equality and its properties. Suitability of the special rewrite system of [25] is demonstrated this chapter of his thesis. The building blocks of de Mol's definition of *semantical equality* between two expressions ($Equal_{\psi}(.,.)$) can be found in Figure 2.24. (For precise definitions consult [23].) Figure 2.24 also contains example de Mol's semantical equalities. SPARKLE currently cannot prove the equalities in Figure 2.24 because of its lack of co-induction. This chapter of de Mol's thesis proves *reducibility* of his semantical equality and reports that its *referential transparency* is also proved in [23]. (In a nutshell, reducibility means that equality is invariant under reduction and referential transparency means that, within a given scope, all occurrences of an expression denote the same value.)

Formal reasoning about general type classes is the main concern in [109]. This is first shown not to be possible in ISABELLE — the only proof assistant that the authors of [109] knew that could support theorem proving about type classes by the time of their writing. Next, they show how they have developed a new inductive scheme which results in a strong and effective proof rule and tactic for assisting proofs about general type classes in SPARKLE.

In [4], a common formal model is first developed for a certain class of desktop and web applications. This is based on a point-free style of Arrow combinators [45] for the *GEC* [5, 6] and *iData* [82, 83] GUI design toolkits. This common formal method is then used for formal reasoning about GUIs developed under both *GEC* and *iData*.

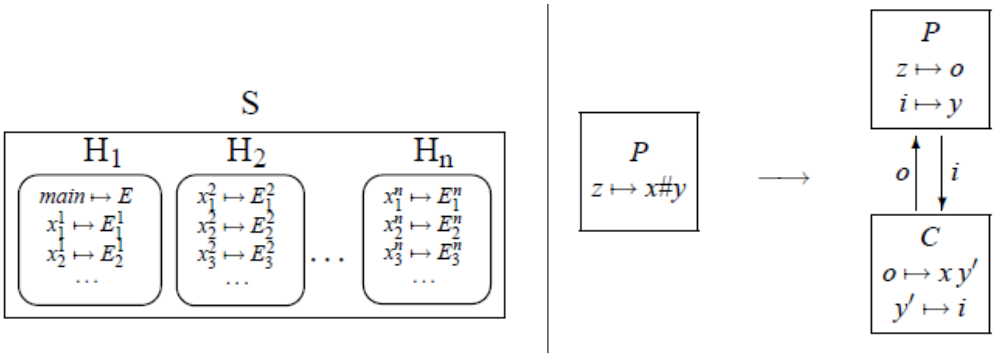


Figure 2.25: Distributed Model of Eden (Left), Process Creation for $x\#y$ (Right)

Sánchez-Gil, Hidalgo-Herrero, and Ortega-Mallén (2009)

Sánchez-Gil, Hidalgo-Herrero, and Ortega-Mallén [92] update the former small-step operational semantics for distributed lazy evaluation [40] which corresponds to the programming language Eden[41]. Their operational semantics is based on Launchbury's natural semantic for lazy evaluation [55] and is very close to the one presented in [11] for GpH. Figure 2.25 (Left) demonstrated the computation model of Eden where H_is are heaps.

The main difference between this model and that of GpH is that, although bindings are unique and shared inside each individual heap, duplication is allowed across heaps. As a matter of fact, in Eden, $x\#y$ is a means for the programmer to trigger *parallel application*. As shown in Figure 2.25 (Right), upon the evaluation of a binding $z \mapsto x\#y$ in a process P, a new process C is created with enough bindings to carry on with the evaluation of the application $x\ y$. The two processes P and C communicate using channels i and o.

The operational semantics of [92] consists of local rules and global rules. Their local rules as well as the global rules that do not consider parallel application correspond closely to those of the operational semantics of GpH (Figure 2.20). Therefore, Figure 2.26 only depicts the rules for $x\#y$. Note the following conventions in Figure 2.26: S represents the rest of the system (consisting of a number of processes); processes are of the form $\langle p, H\rangle$ where p is the process ID and H is its corresponding heap; finally, when $dom(H_1) \cap dom(H_2) = \varnothing$, the notation $H_1 + H_2$ is defined to be $H_1 \cup H_2$, and is undefined otherwise.

Sánchez-Gil, Hidalgo-Herrero, and Ortega-Mallén [92] prove that their operational semantics is correct and computationally adequate with respect to the denotational semantics of Abramsky and Ong [72, 2, 3]. This is done via two intermediate semantics: a one-processor version of their operational semantics, and an extension over Launchbury's operational semantics [55]. They also prove the determinacy[18] of their operational semantics.

[18]Determinacy in this context means that the result of a computation is not dependent on the number of processors.

Syntax

$$E ::= x \mid \lambda x.E \mid x\ y \mid x\#y \mid \mathsf{let}\ \{x_i \mapsto e_i\}_{i=1}^{n}\ \mathsf{in}\ x$$

Parallel Application Rules

(blocking p.c.) $(S, \langle p, H + \{\theta \overset{IA}{\mapsto} x\#y\}\rangle) \overset{bpc}{\rightarrow} (S, \langle p, H + \{\theta \overset{B}{\mapsto} x\#y\}\rangle)$

(proc. creat.) if $\neg\mathsf{d}(x, H + \{\theta \overset{\alpha}{\mapsto} x\#y\})$, q, z, i, o fresh, η fresh renaming

$(S, \langle p, H + \{\theta \overset{\alpha}{\mapsto} x\#y\}\rangle) \overset{pc}{\rightarrow}$

$(S, \langle p, H + \{\theta \overset{B}{\mapsto} o, i \overset{A}{\mapsto} y\}\rangle, \langle q, \eta(\mathsf{nh}(x, H)) + \{o \overset{A}{\mapsto} \eta(x)\ z, z \overset{B}{\mapsto} i\}\rangle)$

(value comm.) if $\neg\mathsf{d}(W, H_p), \eta$ fresh renaming

$(S, \langle p, H_p + \{ch \overset{\alpha}{\mapsto} W\}\rangle, \langle c, H_c + \{\theta \overset{B}{\mapsto} ch\}\rangle) \overset{com}{\rightarrow}$

$(S, \langle p, H_p\rangle, \langle c, H_c + \eta(\mathsf{nh}(W, H_p)) + \{\theta \overset{A}{\mapsto} \eta(W)\}\rangle)$

Auxiliary Functions

$$\mathsf{d}(x, H) = \begin{cases} \mathit{true} & \text{if } \theta \overset{\alpha}{\mapsto} e \in \mathsf{nh}(x, H) \text{ with } (E \equiv ch \vee E \equiv y\#z) \\ \mathit{false} & \text{otherwise} \end{cases}$$

$\mathsf{nh}(E, H)$ is the greatest set of bindings satisfying (i) and (ii)

(i) $x \overset{\alpha}{\mapsto} E \in H \Rightarrow \{x \overset{I}{\mapsto} E\} \cup \mathsf{nh}(E, H) \subseteq \mathsf{nh}(x, H)$

(ii) $\bigcup_{E' \mathsf{subexp}(E)} \mathsf{nh}(E', H) \subseteq \mathsf{nh}(E, H)$

Figure 2.26: Global Rules of [92] Related to Parallel Application.

Summary In this chapter, we first consider the literature on call-by-need in Section 2.1. We justify our design decision in basing our work on that of Launchbury [55]. Next, in Section 2.2, we take a look at the change the semantics of `seq` has experienced over the time. We clarify the `seq` semantics that we choose to study for selective strictness. In Section 2.3, we review the literature on selective strictness which is the main concern of this thesis. We design our operational semantics to be a variation of that of van Eekelen and de Mol [106] which extends that of Launchbury [55]. The details of our reasons for making this design decision is explored in Section 2.3 where we summarise the work of each group with a discussion on how they fit our purposes. Finally, Section 2.4 explores other related work.

Chapter 3

Operational Semantics

In this chapter, we first present an operational semantics in Section 3.1. Next, in Section 3.2, we explore a number of fundamental properties of this operational semantics. Proofs of Section 3.2 are all based on the operational semantics of Section 3.1. The results in Section 3.2 will later be useful in this thesis. In particular, Determinism (Theorem 3.9) is central to our entire system, and Lemmas 3.10 and 3.11 will be useful in Chapters 6 and 7.

3.1 Syntax and Operational Semantics

We start by presenting some syntax (Definition 3.1). We then give a big step operational semantics based on Launchbury's [55], and closely related to [106] (Definition 3.4).

Definition 3.1. *Fix a countably infinite set of* ***variable symbols****. $x, y, z, \ldots$ and $x_1, x_2, x_3, \ldots$ will range over variable symbols.*

Define ***expressions*** *and* ***values*** *by*

$$\begin{array}{lll} e & ::= & x \mid \lambda x.e \mid e\,x \mid \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e \mid e\ \mathsf{seq}\ e \\ v & ::= & \lambda x.e \end{array}$$

e, e', e_1, ... will range over expressions. v, v', v_1, ... will range over values.

We quotient syntax up to α-equivalence of λ- and let*-bound variables. For example, $\lambda x.x = \lambda y.y$. We write $e[y/x]$ for the usual capture-avoiding substitution of x by y in e. For example, $(\lambda y.x)[y/x] = \lambda y'.y$, if x, y, and y' denote distinct variable symbols.*

Definition 3.2. *Define $fv(e)$ the* ***free variables*** *of e by:*

$$fv(x) = \{x\} \quad fv(\lambda x.e) = fv(e) \setminus \{x\} \quad fv(e\,x) = fv(e) \cup \{x\}$$

$$fv(\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e) = (fv(e) \cup \bigcup_{i=1}^{n} fv(e_i)) \setminus \{x_1, \ldots, x_n\}$$

$$fv(e'\ \mathsf{seq}\ e) = fv(e') \cup fv(e)$$

Note that we write $(fv(e) \cup \bigcup_{i=1}^{n} fv(e_i)) \backslash \{x_1, \ldots, x_n\}$ and not $(fv(e) \backslash \{x_1, \ldots, x_n\}) \cup \bigcup_{i=1}^{n} fv(e_i)$. Here, we are following [55] and allow let to express recursion. See Remark 3.5 for further explanation.

Definition 3.3. *If f is a partial function write $dom(f)$ for the* ***domain*** *of f. In symbols, $dom(f) = \{x \mid f(x)$ defined$\}$.*

Call a partial function Γ mapping variable symbols to expressions and such that $dom(\Gamma)$ is finite, a ***heap****. Γ, Δ, Θ, and Ξ will range over heaps.*

Suppose $x \notin dom(\Gamma)$. Define $(\Gamma, x \mapsto e)$ by:

$$\begin{array}{lll} (\Gamma, x \mapsto e)(x) = e & & \\ (\Gamma, x \mapsto e)(y) = \Gamma(y) & & y \in dom(\Gamma) \\ (\Gamma, x \mapsto e)(y) & \textit{undefined} & y \notin dom(\Gamma) \cup \{x\} \end{array}$$

Also write

$$\begin{array}{llll} (\Gamma, x_i \mapsto e_i)_{i=1}^{1} & \textit{for} & (\Gamma, x_1 \mapsto e_1), & \textit{and} \\ (\Gamma, x_i \mapsto e_i)_{i=1}^{n} & \textit{for} & ((\Gamma, x_i \mapsto e_i)_{i=1}^{n-1}, x_n \mapsto e_n). & \end{array}$$

Definition 3.4. *Define an* ***operational semantics*** $\Gamma : e \Downarrow \Delta : v$ *by:*

$$\frac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\ (\mathbf{lam}) \qquad \frac{\Gamma : e \Downarrow \Delta : v}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto v) : v}\ (\mathbf{var_x})$$

$$\frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \quad \Theta : e'[x/y] \Downarrow \Delta : v}{\Gamma : e\, x \Downarrow \Delta : v}\ (\mathbf{app})$$

$$\frac{(\Gamma, x_i \mapsto e_i)_{i=1}^{n} : e \Downarrow (\Delta, x_i \mapsto e_i')_{i=1}^{n} : v \quad (x_i \textit{ fresh}, 1 \leq i \leq n)}{\Gamma : \mathsf{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathsf{in}\ e \Downarrow \Delta : v}\ (\mathbf{let})$$

$$\frac{\Gamma : e_1 \Downarrow \Theta : v_1 \quad \Theta : e_2 \Downarrow \Delta : v_2}{\Gamma : e_1\ \mathsf{seq}\ e_2 \Downarrow \Delta : v_2}\ (\mathbf{seq})$$

In (**var$_\mathbf{x}$**)*, recall that by Definition 3.3 we assume $x \notin dom(\Gamma)$.*

In (**let**) *'x_i fresh' means that $x_i \notin fv(v)$, and $x_i \notin dom(\Gamma)$ and $x_i \notin fv(\Gamma(x))$ for any $x \in dom(\Gamma)$, and similarly for Δ.* [1]

Remark 3.5. Recall that we equate terms up to α-equivalence of let-bound variables. It is convenient to impose a well-formedness condition on derivations, that the new variable names above the line in (**let**) are introduced distinct (so they do not 'clash' with other variable names elsewhere in the derivation). This standard 'trick' is often called Barendregt's naming convention [12].

A particularly interesting effect that this has in our operational semantics is how we define free variables for let expressions (Definition 3.2). For any expression

$$e = \mathsf{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathsf{in}\ e'$$

[1] *Consistent with [55] and subsequent work, we allow the possibility that $x_i \in fv(e_j)$ or $x_i \in fv(e_j')$ for $1 \leq i, j \leq n$.*

to be evaluated, all occurrences of x_i's in e address the local let-bound x_i's. Especially, so are the free occurrences of x_i's which are therefore needed to be distinguished from free variables of e.

We take a moment for a few observations:

- $\lambda x.e$ is the usual functional abstraction. $e\,x$ is the usual function application. This syntax is restricted (it has to be e applied to a *variable symbol*) but this just simplifies our mathematics; expressivity is not lost because we also have let. This is standard; we inherit it from [55].

- We identify syntax up to α-conversion. [55] does not. The $\hat{z}$ in his *Variable* reduction rule [55, Figure 1] makes ($\mathbf{var_x}$) above look different from *Variable*. But $\hat{z}$ is just his notation for 'an α-equivalent term to z'. Thus, Launchbury isolates all the α-conversion needed in his reductions in that single rule. [2] Our design choice is less computational, but we prefer the more abstract representation of syntax.

- We 'garbage-collect' the bindings $x_i \mapsto e'_i$ in the final heap (the Δ) in (**let**), unlike [55, 105]. We will discuss this point in some detail:

  This design choice brings us benefits later. Thanks to 'garbage-collection', let-bindings are not propagated 'outside their scope' to the final heap and so we do not have to reason explicitly 'up to' these choices. (For example: without a garbage-collecting (**let**) rule we would have to 'hard-wire' the same effect into Theorem 8.4 and Corollary 8.6.)

  The rule for let in [55, 105] does not garbage collect. For them, heaps are left with extra bindings after a let-binding is evaluated.

  The semantics in [55, 105] allows variables to escape their scope during evaluation, which is forbidden in our semantics. For example, let $x = \lambda y.y$ in $\lambda z.(xz)$ will evolve in [55] to place a binding $x \mapsto \lambda y.y$ in the heap; this could then be 'accidentally' bound to an x occurring elsewhere, outside the scope of the let. Launchbury is well aware of this issue and comments on it [55, Section 3.1]. To avoid 'accidental name-clash' in evaluations in his system, Launchbury imposes a normalisation procedure which chooses all variable names distinct before evaluation begins.

  This is fine, if we just want to evaluate a particular expression, normalised, in a particular heap. However, for reasoning about the evaluation of classes of programs and proving operational equivalences between them, a garbage-collecting rule for let is better.

  On the other hand, our specific kind of garbage-collection causes restrictions to the expressiveness of our system. For example, the arguably reasonable

[2] Note that in [55], z ranges over values. Also, note that the *normalisation* mentioned in [55, Section 3.1] describes a translation from full λ-calculus syntax to a target syntax which is the restricted, mathematically convenient, language which Launchbury's operational semantics uses; implicit in that procedure is an α-conversion step, but that is in the full λ-calculus syntax and this should not be confused with α-conversion, or α-convertability, in the target syntax.

program $e \equiv \mathsf{let}\ y = \lambda z.z\ \mathsf{in}\ \lambda x.y$ does not reduce in our system. Suppose on the contrary that Π is the derivation $\Gamma : e \Downarrow \Delta : v$ for some heaps Γ, Δ, and value v. Then, Π takes the form

$$\dfrac{\dfrac{}{\Gamma \cup \{y \mapsto \lambda z.z\} : \lambda x.y \Downarrow \Gamma \cup \{y \mapsto \lambda z.z\} : \lambda x.y}\ (\textbf{lam})}{\Gamma : e \Downarrow \Delta : \lambda x.y}\ (\textbf{let})^*.$$

However, this is not correct because the above (**let**)* is not a valid instance of (**let**). The reason is that $y \in fv(\lambda x.y)$ which is not allowed by our operational semantics. See Section 11.3 for more on this.

- The intuition of e_1 seq e_2 is 'evaluate e_1; throw the result away ... but keep the heap and use it to evaluate e_2'. This is very similar to the Clean (**StrictLet**) rule [106, 84]. This operational behaviour of seq is the technical focus of this thesis and further discussion of its behaviour will follow. See Remark 5.2 for example for a more precise explanation of the above intuition. For a discussion on a subtly different selective strictness enforcing mechanisms of Haskell see Section 2.2.

Definition 3.6. *A **derivation** is the labelled tree — labelled with terms, heaps, and derivation rules from Definition 3.4 — that justifies a reduction* $\Gamma : e \Downarrow \Delta : v$. Π, Π_1, Π', Π_x, ... *will range over derivations.*

We may write '$\Gamma : e \Downarrow \Delta : v$' *as shorthand for* "$\Gamma : e \Downarrow \Delta : v$ *is derivable*".

We may write '$\Gamma : e \Downarrow_\Pi \Delta : v$' *as shorthand for* "$\Pi$ *is a derivation of* $\Gamma : e \Downarrow \Delta : v$".

3.2 Fundamental Properties

In this section, we provide a few results which become handy later, and present fundamental properties of our system. In particular, Lemma 3.7 serves proof of determinism (Theorem 3.9) which is of special importance. It is noteworthy that Theorem 3.9 is correct for Launchbury's system [55], as well as that of van Eekelen and de Mol [105, 106, 108].

Lemma 3.7. *If* $\Gamma : e \Downarrow \Delta : v$ *then* $dom(\Gamma) = dom(\Delta)$.

Proof. Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$. We proceed by induction on Π, based on its final rule:

- (**lam**). Π takes the form

$$\dfrac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\ (\textbf{lam})$$

in which case the result is trivial.

- ($\mathbf{var_x}$). Π takes the form

$$\frac{\begin{array}{c}\vdots\\ \Gamma' : e \Downarrow \Delta' : v\end{array}}{(\Gamma', x \mapsto e) : x \Downarrow (\Delta', x \mapsto v) : v}\ (\mathbf{var_x})$$

  where $\Gamma = (\Gamma', x \mapsto e)$ and $\Delta = (\Delta', x \mapsto v)$. By inductive hypothesis, $dom(\Gamma') = dom(\Delta')$. The result follows by noting that $dom(\Gamma) = dom(\Gamma') \cup \{x\}$ and $dom(\Delta) = dom(\Delta') \cup \{x\}$.

- ($\mathbf{app}$). Π takes the form

$$\frac{\begin{array}{cc}\vdots & \vdots\\ \Gamma : e \Downarrow \Theta : \lambda y.e' & \Theta : e'[x/y] \Downarrow \Delta : v\end{array}}{\Gamma : e\ x \Downarrow \Delta : v}\ (\mathbf{app}).$$

  By inductive hypothesis, $dom(\Gamma) = dom(\Theta)$. Similarly, by inductive hypothesis, $dom(\Theta) = dom(\Delta)$. The result follows by transitivity.

- ($\mathbf{let}$). Π takes the form

$$\frac{\begin{array}{c}\vdots\\ (\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow (\Delta, x_i \mapsto e_i')_{i=1}^n : v\end{array}}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : v}\ (\mathbf{let}).$$

  By inductive hypothesis, $dom((\Gamma, x_i \mapsto e_i)_{i=1}^n) = dom((\Delta, x_i \mapsto e_i')_{i=1}^n)$. The result follows by noting that $dom((\Gamma, x_i \mapsto e_i)_{i=1}^n) = dom(\Gamma) \cup \{x_i\}_{i=1}^n$ and $dom((\Delta, x_i \mapsto e_i)_{i=1}^n) = dom(\Delta) \cup \{x_i\}_{i=1}^n$.

- ($\mathbf{seq}$). Π takes the form

$$\frac{\begin{array}{cc}\vdots & \vdots\\ \Gamma : e_1 \Downarrow \Theta : v_1 & \Theta : e_2 \Downarrow \Delta : v_2\end{array}}{\Gamma : e_1\ \mathsf{seq}\ e_2 \Downarrow \Delta : v_2}\ (\mathbf{seq}).$$

  By inductive hypothesis, $dom(\Gamma) = dom(\Theta)$. Similarly, by inductive hypothesis, $dom(\Theta) = dom(\Delta)$. The result follows by transitivity.

□

Notation 3.8. *Write* $\Pi =_{\mathsf{let}\alpha} \Pi'$ *when* Π *and* Π' *are equal up to renaming* let*-bound variables.*

The following theorem shows that derivations are unique up to an easy renaming, and reduction is deterministic:

Theorem 3.9. *If* $\Gamma : e \Downarrow_\Pi \Delta : v$ *and* $\Gamma : e \Downarrow_{\Pi'} \Delta' : v'$ *then* $\Pi =_{\mathsf{let}\alpha} \Pi'$, $\Delta = \Delta'$, *and* $v = v'$.

Proof. Induction on Π based on its final rule:

- (**lam**). Trivial.
- (**var$_x$**). Let $\Gamma = (\Gamma', x \mapsto e)$, $\Delta_1 = (\Delta_1', x \mapsto v_1)$, and $\Delta_2 = (\Delta_2', x \mapsto v_2)$. Suppose that $\Gamma : x \Downarrow_{\Pi_1} \Delta_1 : v_1$ and $\Gamma : x \Downarrow_{\Pi_2} \Delta_2 : v_2$. Accordingly, Π_1' and Π_2' exist such that $\Gamma' : e \Downarrow_{\Pi_1'} \Delta_1' : v_1$ and $\Gamma' : e \Downarrow_{\Pi_2'} \Delta_2' : v_2$. By inductive hypothesis then $v_1 = v_2$, $\Delta_1' = \Delta_2'$, and $\Pi_1' =_{\mathsf{let}\alpha} \Pi_2'$. The result follows.
- (**app**). Suppose that
  - $\Gamma : e \Downarrow_{\Pi_1} \Theta : \lambda y.e$ and $\Theta : e[x/y] \Downarrow_{\Pi_2} \Delta : v$ and
  - $\Gamma : e \Downarrow_{\Pi_1'} \Theta' : \lambda y.e'$ and $\Theta' : e'[x/y] \Downarrow_{\Pi_2'} \Delta' : v'$

  (we α-convert the bound variable y so it is the same in both derivations). By inductive hypothesis $\Theta = \Theta'$ and $\lambda y.e = \lambda y.e'$. Thus $e[x/y] = e'[x/y]$. We use the inductive hypothesis. On the other hand, by inductive hypothesis, $\Pi_1 =_{\mathsf{let}\alpha} \Pi_1'$. Likewise, $\Pi_2 =_{\mathsf{let}\alpha} \Pi_2'$. The result follows.
- (**let**). Suppose that

$$(\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow_{\Pi_1} (\Delta, x_i \mapsto e_{1i}')_{i=1}^n : v, \quad \text{and}$$
$$(\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow_{\Pi_1'} (\Delta', x_i \mapsto e_{2i}')_{i=1}^n : v'.$$

  By inductive hypothesis $v = v'$ and $(\Delta, x_i \mapsto e_{1i}')_{i=1}^n = (\Delta', x_i \mapsto e_{2i}')_{i=1}^n$. It follows easily that $\Delta = \Delta'$. By the other part of inductive hypothesis, $\Pi_1 =_{\mathsf{let}\alpha} \Pi_1'$, which implies $\Pi =_{\mathsf{let}\alpha} \Pi'$.
- (**seq**). Suppose that
  - $\Gamma : e_1 \Downarrow_{\Pi_1} \Theta : v_1$ and $\Theta : e_2 \Downarrow_{\Pi_2} \Delta : v_2$ and
  - $\Gamma : e_1 \Downarrow_{\Pi_1'} \Theta' : v_1'$ and $\Theta' : e_2 \Downarrow_{\Pi_2'} \Delta' : v_2'$

  By inductive hypothesis, $\Theta = \Theta'$ and $\Pi_1 =_{\mathsf{let}\alpha} \Pi_1'$. Thus, by inductive hypothesis, $\Delta = \Delta'$, $\Pi_2 =_{\mathsf{let}\alpha} \Pi_2'$, and $v_2 = v_2'$. The result is immediate.

□

Lemma 3.10. *Suppose that $\Gamma(x)$ is a value v_x and $\Gamma : e \Downarrow_\Pi \Delta : v$. Then, for every heap Ξ in Π such that $x \in dom(\Xi)$, $\Xi(x) = v_x$. In particular, $\Delta(x) = v_x$.*

Proof. Let $\Gamma : e \Downarrow_\Pi \Delta : v$ where $\Gamma(x)$ is a value v_x. We proceed by induction on Π based on its final rule:

- (**var$_y$**) for y other than x. Π takes the form

$$\frac{\begin{array}{c}\vdots\,\Pi'\\ \Gamma' : e_y \Downarrow \Delta' : v_y\end{array}}{(\Gamma', y \mapsto e_y) : y \Downarrow (\Delta', y \mapsto v_y) : v_y}\,(\mathbf{var_y})$$

where $(\Gamma', y \mapsto e_y) = \Gamma$ and $(\Delta', y \mapsto v_y) = \Delta$. Given that $\Gamma'(x) = \Gamma(x) = v_x$, by inductive hypothesis, for every heap Ξ in Π' such that $x \in dom(\Xi)$, $\Xi(x) = v_x$, and in particular $\Delta'(x) = v_x$. The result follows by noting that $\Delta(x) = \Delta'(x)$.

There is nothing to prove for the cases (**lam**) (because $\Gamma = \Delta$) and ($\mathbf{var_x}$) (because $x \notin dom(\Xi)$ for every Ξ in Π except Γ and Δ). The case (**let**) is similar to ($\mathbf{var_y}$). The cases (**app**) and (**seq**) follow immediately by transitivity. □

Lemma 3.11. *Suppose that* $\Gamma : e \Downarrow \Delta : v$ *and* $\Gamma(x) \neq \Delta(x)$ *for some* $x \in dom(\Gamma)$. *Then,* $\Delta(x)$ *is a value.*

Proof. Induction on Π based on its final rule:

- ($\mathbf{var_x}$). Π takes the form

$$\frac{\begin{array}{c}\vdots\,\Pi' \\ \Gamma' : e_x \Downarrow \Delta' : v_x\end{array}}{(\Gamma', x \mapsto e_x) : x \Downarrow (\Delta', x \mapsto v_x) : v_x}\ (\mathbf{var_x}).$$

  By definition, $v_x = \Delta(x)$ is a value.

- ($\mathbf{var_y}$) for y other than x. Π takes the form

$$\frac{\begin{array}{c}\vdots\,\Pi' \\ \Gamma' : e_y \Downarrow \Delta' : v_y\end{array}}{(\Gamma', y \mapsto e_y) : y \Downarrow (\Delta', y \mapsto v_y) : v_y}\ (\mathbf{var_y})$$

  where $(\Gamma', y \mapsto e_y) = \Gamma$ and $(\Delta', y \mapsto v_y) = \Delta$. If $\Gamma(x) \neq \Delta(x)$, then $\Gamma'(x) = \Delta'(x)$ because $\Gamma(x) = \Gamma'(x)$ and $\Delta(x) = \Delta'(x)$. By inductive hypothesis, we conclude that $\Gamma'(x)$ is a value. The result follows by noting that $\Delta'(x) = \Delta(x)$.

- (**app**). Π takes the form

$$\frac{\begin{array}{cc}\vdots\,\Pi_1 & \vdots\,\Pi_2 \\ \Gamma : e \Downarrow \Theta : \lambda z.e' & \Theta : e'[y/z] \Downarrow \Delta : v\end{array}}{\Gamma : e\ y \Downarrow \Delta : v}\ (\mathbf{app}).$$

  We now consider cases:

  - $\Gamma(x) = \Theta(x)$. By $\Gamma(x) \neq \Delta(x)$, it follows that $\Theta(x) \neq \Delta(x)$. By inductive hypothesis, $\Delta(x)$ is a value, as expected.
  - $\Gamma(x) \neq \Theta(x)$. By inductive hypothesis, then, $\Theta(x)$ is a value like v_x. By Lemma 3.10, $\Delta(x) = v_x$. The result follows.

There is nothing to prove in the (**lam**) case (because $\Gamma = \Delta$). The (**let**) and (**seq**) cases are similar to ($\mathbf{var_y}$) and (**app**), respectively. □

Chapter 4

Notions of Equivalence

Three plausible notions of equivalence between expressions offer themselves: *value* equivalence $\approx_v$, *heap* equivalence $\approx_h$, and *strict* equivalence $\approx_s$ (defined below). While it is easy to define notions of equivalence, what makes them interesting is the theorems we prove about them. We work with $\approx_s$, as the strongest equivalence offering the most scope for proving strong properties. Later, we prove the non-trivial fact that — in a Launchbury-based system like ours that the **(let)** rule garbage-collects — $\approx_v$ coincides with $\approx_s$ (Section 8.2).

4.1 Definitions

In this section, we define three notions of equivalence between expressions. It is noteworthy that, although we use them for our own operational semantics here, they are perfectly definable for any Launchbury-based operational semantics.

Definition 4.1. *Define* $e_1 \approx_v e_2$ *by:*

$$\forall \Gamma, v.\ (\exists \Delta_1.\ \Gamma : e_1 \Downarrow \Delta_1 : v) \Leftrightarrow (\exists \Delta_2.\ \Gamma : e_2 \Downarrow \Delta_2 : v)$$

We call e_1 *and* e_2 ***value equivalent****.*

Intuitively $e_1 \approx_v e_2$ when e_1 and e_2 compute the same final value, given the same initial heap. For example: $\mathsf{let}\ \{x = \lambda x.x, y = \lambda x.x\}\ \mathsf{in}\ x\ y \approx_v \lambda x.x$.

Definition 4.2. *Define* $e_1 \approx_h e_2$ *by:*

$$\forall \Gamma, \Delta.\ (\exists v_1.\ \Gamma : e_1 \Downarrow \Delta : v_1) \Leftrightarrow (\exists v_2.\ \Gamma : e_2 \Downarrow \Delta : v_2)$$

We call e_1 *and* e_2 ***heap equivalent****.*

Intuitively, $e_1 \approx_h e_2$ when e_1 and e_2 compute the same final heaps, given the same initial heap; we do not examine the final values, which may differ, or the order of evaluation. For example, $x\ \mathsf{seq}\ y \approx_h y\ \mathsf{seq}\ x$ (this follows by Corollary 9.2) and it is easy to show that $x\ \mathsf{seq}\ y \not\approx_v y\ \mathsf{seq}\ x$.

Definition 4.3. *Define* $e_1 \cong_s e_2$ *by:*

$$\forall \Gamma, v, \Delta.\ (\Gamma : e_1 \Downarrow \Delta : v) \Leftrightarrow (\Gamma : e_2 \Downarrow \Delta : v)$$

We call e_1 *and* e_2 ***strictly equivalent****.*

Intuitively, $e_1 \cong_s e_2$ when, given the same initial heap, e_1 and e_2 compute the same final value and final heap. For example $(\lambda x.x)y \cong_s y$, and $x \ \mathsf{seq}\ y \ncong_s y$. Less obviously, $x_1 \ \mathsf{seq}\ (x_2 \ \mathsf{seq}\ x_3) \cong_s x_2 \ \mathsf{seq}\ (x_1 \ \mathsf{seq}\ x_3)$ (from Theorems 5.1 and 9.3).

4.2 Observations about Notions of Equivalence

Lemma 4.4. $\cong_v$, $\cong_h$, *and* $\cong_s$ *are equivalence relations (reflexive, symmetric, transitive).*

Proof. We prove the reult for $\cong_s$. The proof is similar for $\cong_h$ and $\cong_v$.

- Reflexivity. For any expression e, the following statement is a tautology which means that $e \cong_s e$:

$$\forall \Gamma, v, \Delta.\ (\Gamma : e \Downarrow \Delta : v) \Leftrightarrow (\Gamma : e \Downarrow \Delta : v).$$

- Symmetry. Fix expressions e_1 and e_2 such that $e_1 \cong_s e_2$. By Definition 4.3, this means that:

$$\forall \Gamma, v, \Delta.\ (\Gamma : e_1 \Downarrow \Delta : v) \Leftrightarrow (\Gamma : e_2 \Downarrow \Delta : v).$$

  Hence,

$$\forall \Gamma, v, \Delta.\ (\Gamma : e_2 \Downarrow \Delta : v) \Leftrightarrow (\Gamma : e_1 \Downarrow \Delta : v).$$

  Namely, $e_2 \cong_s e_1$.

- Transitivity. Fix expressions e_1, e_2, and e_3 such that $e_1 \cong_s e_2$ and $e_2 \cong_s e_3$. Furthermore, fix arbitrary Γ_0, Δ_0, and v_0 for which $\Gamma_0 : e_1 \Downarrow \Delta_0 : v_0$. By Definition 4.3, from $e_1 \cong_s e_2$, we conclude $\Gamma_0 : e_2 \Downarrow \Delta_0 : v_0$. Therefore, again by Definition 4.3, from $e_2 \cong_s e_3$, we conclude $\Gamma_0 : e_3 \Downarrow \Delta_0 : v_0$, as required.

□

Lemma 4.5. $e_1 \cong_s e_2$ *if and only if* $e_1 \cong_h e_2$ *and* $e_1 \cong_v e_2$.

Proof. The left-to-right direction is obvious. Suppose that $e_1 \cong_v e_2$ and $e_1 \cong_h e_2$. Fix arbitrary Γ, Δ_1, and v_1 such that

$$\Gamma : e_1 \Downarrow \Delta_1 : v_1. \tag{4.1}$$

Because $e_1 \cong_v e_2$, Definition 4.1 and (4.1) imply that there exists a heap Δ_2 such that

$$\Gamma : e_2 \Downarrow \Delta_2 : v_1. \tag{4.2}$$

On the other hand, because $e_1 \cong_h e_2$, Definition 4.1 and (4.1) imply that there exists a value v_2 such that

$$\Gamma : e_2 \Downarrow \Delta_1 : v_2. \tag{4.3}$$

Finally, because our operational semantics is deterministic (Theorem 3.9), (4.2) and (4.3) imply that $v_1 = v_2$ and $\Delta_1 = \Delta_2$. □

Theorem 4.6 formalises an intuition that seq evaluates its first argument, then 'throws it away and keeps the heap':

Theorem 4.6. *$e_1 \cong_h e_1'$ and $e_2 \cong_s e_2'$ imply e_1 seq $e_2 \cong_s e_1'$ seq e_2'.*

Proof. Suppose $\Gamma : e_1 \text{ seq } e_2 \Downarrow_\Pi \Delta : v$. Π takes the form

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma : e_1 \Downarrow \Theta : v_1 & \Theta : e_2 \Downarrow \Delta : v\end{array}}{\Gamma : e_1 \text{ seq } e_2 \Downarrow \Delta : v}\ (\textbf{seq}).$$

By assumption $e_1 \cong_h e_1'$, so $\Gamma : e_1' \Downarrow \Theta : v_1'$ for some v_1'. By assumption $e_2 \cong_s e_2'$, so $\Theta : e_2' \Downarrow \Delta : v$. We construct a derivation as follows

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma : e_1' \Downarrow \Theta : v_1' & \Theta : e_2' \Downarrow \Delta : v\end{array}}{\Gamma : e_1' \text{ seq } e_2' \Downarrow \Delta : v}\ (\textbf{seq}).$$

The result follows by symmetry. □

A converse to Theorem 4.6 is false; see Corollary 5.6.

$\cong_s$ is a congruence with respect to seq and to application:

Theorem 4.7. *1. $e_1 \cong_s e_1'$ and $e_2 \cong_s e_2'$ imply e_1 seq $e_1' \cong_s e_2$ seq e_2'.*

2. $e \cong_s e'$ implies $e\ x \cong_s e' x$.

3. $e \cong_s e'$ implies

$$\text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e \cong_s \text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e'.$$

Proof. 1. From Lemma 4.5 and Theorem 4.6.

2. Suppose $\Gamma : e\ x \Downarrow_\Pi \Delta : v$. Π takes the form

$$\frac{\begin{array}{cc}\vdots\ \Pi_1 & \vdots\ \Pi_2 \\ \Gamma : e \Downarrow \Theta : \lambda z.e'' & \Theta : e''[x/z] \Downarrow \Delta : v\end{array}}{\Gamma : e\ x \Downarrow \Delta : v}\ (\textbf{app}).$$

By assumption $\Gamma : e' \Downarrow \Theta : \lambda z.e''$. We combine this with Π_2 to observe that $\Gamma : e'\ x \Downarrow \Delta : v$ as required.

3. Suppose $e \cong_s e'$. Suppose $\Gamma : \mathsf{let}\ \{x_i = e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow_\Pi \Delta : v$. Π takes the form

$$\dfrac{\begin{array}{c}\vdots\ \Pi_1 \\ (\Gamma, x_i{\mapsto}e_i)_{i=1}^n : e \Downarrow (\Delta, x_i{\mapsto}e_i'')_{i=1}^n : v\end{array}}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : v}\ (\mathbf{let}).$$

Because $e \cong_s e'$, there exists a Π_1' such that:

$$\dfrac{\begin{array}{c}\vdots\ \Pi_1' \\ (\Gamma, x_i{\mapsto}e_i)_{i=1}^n : e' \Downarrow (\Delta, x_i{\mapsto}e_i'')_{i=1}^n : v\end{array}}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e' \Downarrow \Delta : v}\ (\mathbf{let}).$$

The result follows. □

Remark 4.8. We hypothesise that also, $e \cong_s e'$ and $e_i \cong_s e_i'$ for $1 \leq i \leq n$ implies

$$\mathsf{let}\ \{x_i = e_i\}_{i=1}^n\ \mathsf{in}\ e \cong_s \mathsf{let}\ \{x_i = e_i'\}_{i=1}^n\ \mathsf{in}\ e'.$$

The proof would probably use Theorem 8.11. Considering this is future work.

Note that $\cong_s$ is not a congruence for λ. For example, $x\ \mathsf{seq}\ x \cong_s x$ (Theorem 5.5) and $\lambda x.(x\ \mathsf{seq}\ x) \ncong_s \lambda x.x$. This is because $\cong_s$ is intensional on values; $v \cong_s v'$ when v and v' are equal up to α-equivalence.

In fact, this has a semantic analogue in the use of *valuations* in denotationally-based techniques like that of [106]; valuations make terms 'closed', by assigning meanings to all their free variables. It is possible to consider extensional notions of operational equivalence (see [81] for an overview), and this is for future work. However, our intensional equivalences are already interesting, and indeed, they probably provide a less cluttered canvas with which to illustrate the proof-methods which we develop for seq.

Chapter 5

Elementary Properties of seq

This chapter presents some elementary properties of seq. Although they are elementary, perhaps they can still be quite surprising.

seq is called 'sequential composition', so we start by proving a typical sequential property, that seq is associative (Theorem 5.1). In Remark 5.2 we discuss a subtlety of the interpretation of seq, that the intuition of e_1 seq e_2 as 'e_1 then e_2' can mislead: e_1 can trigger some (or even all) of the evaluations in e_2, so that e_2 is still evaluated 'first'. Finally, we prove that seq is idempotent (Theorem 5.5). The result we give in this chapter is restricted to variable symbols; the proof for general expressions is rather harder, and comes later on in Chapter 10. Theorem 5.5 is still useful, to establish a counterexample in Corollary 5.6.

Theorem 5.1 (Associativity)**.**
e_1 seq $(e_2$ seq $e_3) \cong_s (e_1$ seq $e_2)$ seq e_3

Proof. Suppose $\Gamma : e_1$ seq $(e_2$ seq $e_3) \Downarrow_\Pi \Delta : v_3$. Π takes the form

$$\dfrac{\Gamma : e_1 \Downarrow \Theta : v_1 \quad \dfrac{\Theta : e_2 \Downarrow \Xi : v_2 \quad \Xi : e_3 \Downarrow \Delta : v_3}{\Theta : e_2 \text{ seq } e_3 \Downarrow \Delta : v_3}\,(\textbf{seq})}{\Gamma : e_1 \text{ seq } (e_2 \text{ seq } e_3) \Downarrow \Delta : v_3}\,(\textbf{seq}).$$

We rearrange this as follows:

$$\dfrac{\dfrac{\Gamma : e_1 \Downarrow \Theta : v_1 \quad \Theta : e_2 \Downarrow \Xi : v_2}{\Gamma : e_1 \text{ seq } e_2 \Downarrow \Xi : v_2}\,(\textbf{seq}) \quad \Xi : e_3 \Downarrow \Delta : v_3}{\Gamma : (e_1 \text{ seq } e_2) \text{ seq } e_3 \Downarrow \Delta : v_3}\,(\textbf{seq}).$$

So $\Gamma : (e_1$ seq $e_2)$ seq $e_3 \Downarrow_\Pi \Delta : v_3$. The reverse implication is similar. □

Remark 5.2. seq is so called because it is 'sequential'. So, one may think of e_1 seq e_2 as 'calculate e_1, then e_2'. But, there are some subtleties. For example,

it is a fact that

$$(x \mapsto (\lambda z.z)y,\ y \mapsto \mathsf{let}\ z = \lambda z.z\ \mathsf{in}\ zz) : x\ \mathsf{seq}\ y \Downarrow_\Pi (x \mapsto \lambda z.z, y \mapsto \lambda z.z) : \lambda z.z.$$

The reader can verify that inside Π, the evaluation of $\mathsf{let}\ z = \lambda z.z\ \mathsf{in}\ zz$, bound to y, is carried out in some sense 'before' that of $(\lambda z.z)y$, bound to x.

$e_1\ \mathsf{seq}\ e_2$ means what **(seq)** in Definition 3.4 says that it means: "evaluate e_1 first, then evaluate e_2 in the resulting heap". It could be that, after evaluating e_1, every evaluation that would have been triggered by evaluating e_2 in the original heap (that is, by evaluating e_2 first) has already been carried out. We return to this in the discussion opening Chapter 7.

It is easy to prove (a restricted form of) idempotence of seq up to $\approx_s$. Lemmas 5.3 and 5.4 are convenient:

Lemma 5.3. *If* $\Gamma : x \Downarrow \Delta : v$ *is derivable then* $\Delta(x) = v$.

Proof. Suppose $\Gamma : x \Downarrow_\Pi \Delta : v$. Π must take the following form where $(\Gamma', x \mapsto e) = \Gamma$ and $(\Delta', x \mapsto v) = \Delta$

$$\frac{\begin{array}{c}\vdots\ \Pi' \\ \Gamma' : e \Downarrow \Delta' : v\end{array}}{(\Gamma', x \mapsto e) : x \Downarrow (\Delta', x \mapsto v) : v}\ (\mathbf{var_x}).$$

Obviously, $\Delta(x) = (\Delta', x \mapsto v)(x) = v$. □

Lemma 5.4. *If* $\Gamma(x) = v$ *then* $\Gamma : x \Downarrow \Gamma : v$.

Proof. Let Γ' be such that $\Gamma = (\Gamma', x \mapsto v)$. Then,

$$\cfrac{\cfrac{}{\Gamma' : v \Downarrow \Gamma' : v}\ (\mathbf{lam})}{\Gamma : x \Downarrow \Gamma : v}\ (\mathbf{var_x})$$

is a valid derivation, and the result follows. □

Theorem 5.5. $x\ \mathsf{seq}\ x \approx_s x$.

Proof. Suppose $\Gamma : x\ \mathsf{seq}\ x \Downarrow_\Pi \Delta : v$. Π takes the form

$$\frac{\begin{array}{cc}\vdots\ \Pi_1 & \vdots\ \Pi_2 \\ \Gamma : x \Downarrow \Theta : v' & \Theta : x \Downarrow \Delta : v\end{array}}{\Gamma : x\ \mathsf{seq}\ x \Downarrow \Delta : v}\ (\mathbf{seq}).$$

By Lemma 5.3 $\Theta(x) = v'$. By Lemma 5.4 $\Theta : x \Downarrow \Theta : v'$. By Theorem 3.9 it follows that $\Delta = \Theta$ and $v' = v$. So Π_1 is a derivation of $\Gamma : x \Downarrow \Delta : v$. □

It is also true that e seq $e \cong_s e$, but the proof is harder. We need to know that $\Gamma : e \Downarrow \Delta : v$ implies $\Delta : e \Downarrow \Delta : v$, that is, re-evaluation of e does not change the heap further (and returns the same value). See Theorems 10.1 and 10.2.

Theorem 5.5 is enough to prove a converse to Theorem 4.6 false:

Corollary 5.6. *Suppose that* e_1' seq $e_1 \cong_s e_2'$ seq e_2. *Then* $e_1 \cong_s e_2$ *does not imply* $e_1' \cong_h e_2'$.

Proof. It suffices to provide a counterexample. By associativity (Theorem 5.1) $(x \text{ seq } z) \text{ seq } z \cong_s x \text{ seq } (z \text{ seq } z)$. By Theorem 5.5 and part 1 of Theorem 4.7 $x \text{ seq } (z \text{ seq } z) \cong_s x \text{ seq } z$. By Lemma 4.4, $(x \text{ seq } z) \text{ seq } z \cong_s x \text{ seq } z$. Also, by Lemma 4.4 $z \cong_s z$. It is not hard to verify that $x \text{ seq } z \ncong_h x$. □

Chapter 6

Further Properties of Launchbury's Semantics

The operational semantics from Definition 3.4 is inherited from [55, 106], with slight modifications. In Chapter 7, we apply some new proof-techniques to that semantics, as discussed in the Introduction. Before we do so, however, it will be useful to make some fundamental observations about the operational semantics. As far as we know, the only place in which these have previously appeared in the literature is in [71]. This thesis borrows largely from [71], where for this particular chapter, all the proofs of the former work are completed.

We distinguish between uses of a variable that cause evaluation *and change the heap* and those that do not *and 'look up' a value, without changing the heap.* This is the distinction between *non-trivial* and *trivial* instances of $(\mathbf{var_x})$ in Definition 6.1. Thus, in Lemma 6.7, we establish that 'Π contains a non-trivial instance of $(\mathbf{var_x})$' and 'Π changes the expression bound to x', are equivalent. We write $\mathit{diff}(\Pi)$ for the set of these variables (Definition 6.4). This notion is important: in Chapters 8 and 9 we will obtain our central results by induction, not on derivations Π, but by induction on the sets $\mathit{diff}(\Pi)$.

The operational semantics is intended to capture call by need evaluation. Thus, we expect that evaluation is 'shared' and that expressions should be evaluated at most once. This intuition is captured by Theorem 6.9. Finally, the only variables which 'matter' to Π are the variables x such that $(\mathbf{var_x})$ appears in Π; other x do not matter. Lemmas 6.15, 6.22, and Theorem 6.23 make that formal.

This chapter is divided into two sections: Section 6.1 explores the differences between instances of $(\mathbf{var_x})$. Section 6.2 classifies the essential parts of a heap for a derivation. Although many of the proofs in this chapter are long, they are mostly routine inductions. Exceptions are Theorem 6.9, and Theorem 6.23.

6.1 Trivial/Non-trivial Instances

In this section, we divide $(\mathbf{var_x})$ instances into two groups: Trivial and Non-Trivial (Definition 6.1). We then formalise our intuition that trivial instances do not change the heap (Lemma 6.5) whilst non-trivial ones do (Lemma 6.7). These

results are then used to prove that each derivation contains at most one ($\mathbf{var_x}$) instance (Theorem 6.9).

Definition 6.1. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$. *Write* $V(\Pi)$ *for the set of* $x \in dom(\Gamma)$ *such that* Π *contains an instance of* ($\mathbf{var_x}$).

Suppose Π *contains an instance of* ($\mathbf{var_x}$), *as illustrated:*

$$\dfrac{\begin{array}{c}\vdots\ \Pi' \\ \Gamma' : e_x \Downarrow \Gamma' : v_x\end{array}}{(\Gamma', x \mapsto e_x) : x \Downarrow (\Gamma', x \mapsto v_x) : v_x}\ (\mathbf{var_x}).$$

Call the instance ***trivial*** *when* Π' *consists of a single instance of* (**lam**) *(equivalently, when* $e_x = v_x$*). Call the instance* ***non-trivial*** *otherwise.*

Lemma 6.2. *Suppose* $\Pi_1 =_{\mathsf{let}\alpha} \Pi_2$. *Then,* $V(\Pi_1) = V(\Pi_2)$.

Proof. Immediate from Notation 3.8 and Definition 6.1. □

Lemma 6.3. *Let* $x \in dom(\Gamma)$ *and* $\Gamma : e \Downarrow_\Pi \Delta : v$. *Then,* Π *contains no non-trivial instance of* ($\mathbf{var_x}$) *if and only if* $\Gamma(x) = \Delta(x)$.

Proof. Let $\Gamma : e \Downarrow_\Pi \Delta : v$ such that Π contains no non-trivial instance of ($\mathbf{var_x}$). We proceed by induction on Π based on its final rule:

- (**lam**). Π takes the form

  $$\dfrac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\ (\mathbf{lam})$$

  in which case the result is trivially correct because $\Gamma = \Delta$ and Π contains no non-trivial instance of ($\mathbf{var_x}$).

- ($\mathbf{var_x}$). Π takes the form

  $$\dfrac{\begin{array}{c}\vdots \\ \Gamma' : e_x \Downarrow \Delta' : v_x\end{array}}{(\Gamma', x \mapsto e_x) : x \Downarrow (\Delta', x \mapsto v_x) : v_x}\ (\mathbf{var_x})$$

  where $(\Gamma', x \mapsto e_x) = \Gamma$ and $(\Delta', x \mapsto v_x) = \Delta$. Given that $x \notin dom(\Gamma')$, this is the only ($\mathbf{var_x}$) instance of Π. And, in order for this very instance to be trivial (Definition 6.1), we must have $e_x = v_x$, as desired. On the other hand, if $e_x = v_x$, then (by Definition 6.1) this is not a non-trivial instance of ($\mathbf{var_x}$).

- ($\mathbf{var_y}$) for y other than x. Π takes the form

  $$\dfrac{\begin{array}{c}\vdots\ \Pi' \\ \Gamma' : e_y \Downarrow \Delta' : v_y\end{array}}{(\Gamma', y \mapsto e_y) : y \Downarrow (\Delta', y \mapsto v_y) : v_y}\ (\mathbf{var_y})$$

where $(\Gamma', y \mapsto e_y) = \Gamma$ and $(\Delta', y \mapsto v_y) = \Delta$. By assumption, Π contains no non-trivial instances of ($\mathbf{var_x}$). Therefore, Π' contains no non-trivial instances of ($\mathbf{var_x}$) either. By inductive hypothesis then $\Gamma'(x) = \Delta'(x)$. The result follows by noting that $\Gamma(x) = \Gamma'(x)$ and $\Delta(x) = \Delta'(x)$.

On the other hand, if $\Gamma(x) = \Delta(x)$, then $\Gamma'(x) = \Delta'(x)$. By inductive hypothesis, Π' contains no non-trivial instance of ($\mathbf{var_x}$). Thus, Π contains no non-trivial instance of ($\mathbf{var_x}$) either.

- ($\mathbf{app}$). By inductive hypothesis.
- ($\mathbf{let}$). Π takes the form

$$\dfrac{\begin{array}{c}\vdots\,\Pi'\\ (\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow (\Delta, x_i \mapsto e_i')_{i=1}^n : v\end{array}}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : v}\ (\mathbf{let}).$$

Given that there are no non-trivial instances of ($\mathbf{var_x}$) in Π, no non-trivial instances of ($\mathbf{var_x}$) exist in Π' either. By inductive hypothesis then $(\Gamma, x_i \mapsto e_i)_{i=1}^n(x) = (\Delta, x_i \mapsto e_i')_{i=1}^n(x)$. The result follows by noting that $(\Gamma, x_i \mapsto e_i)_{i=1}^n(x) = \Gamma(x)$ and $(\Delta, x_i \mapsto e_i')(x) = \Delta(x)$.

On the other hand, if $\Gamma(x) = \Delta(x)$, then $(\Gamma, x_i \mapsto e_i)_{i=1}^n(x) = (\Delta, x_i \mapsto e_i')_{i=1}^n(x)$. By inductive hypothesis, Π' contains no non-trivial instance of ($\mathbf{var_x}$). Thus, Π contains no non-trivial instance of ($\mathbf{var_x}$) either.

- ($\mathbf{seq}$). By inductive hypothesis.

□

Definition 6.4. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$. *Define* $\mathit{diff}(\Pi)$ *by:*

$$\mathit{diff}(\Pi) = \{x \in \mathit{dom}(\Gamma) \mid \Gamma(x) \neq \Delta(x)\}$$

Lemma 6.5. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$. *Then the following are equivalent:*

(1) $x \in V(\Pi)$ *and* $x \notin \mathit{diff}(\Pi)$.

(2) $x \in V(\Pi)$ *and* $\Gamma(x)$ *is a value (and since* $\Delta(x) = \Gamma(x)$*, so is* $\Delta(x)$*).*

Proof. By induction on Π based on its final rule. For each case, we prove that (1) $\Leftrightarrow$ (2):

- ($\mathbf{lam}$). Π takes the form

$$\dfrac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\ (\mathbf{lam}).$$

In this case, both directions are correct because $V(\Pi) = \varnothing$.

- ($\mathbf{var_x}$). Π takes the form

$$\begin{array}{c}\vdots\,\Pi\\ \Gamma : x \Downarrow \Delta : v_x.\end{array}$$

  If $x \notin \mathit{diff}(\Pi)$, by Definition 6.4, $\Gamma(x) = \Delta(x) = v_x$, i.e., $\Gamma(x)$ is a value. (Note that, by construction of our ($\mathbf{var_x}$) rule, we know that $\Delta = (\Delta', x \mapsto v_x)$ for some heap Δ'.) Suppose that $\Gamma(x)$ is a value like v. On one hand, by Lemma 3.10, $\Delta(x) = v$ too. On the other hand, by Lemma 5.3, $\Delta(x) = v_x$. Therefore, by Determinism (Theorem 3.9), $v = v_x$, and $x \notin \mathit{diff}(\Pi)$.

- ($\mathbf{var_y}$) for y other than x. Π takes the form

$$\frac{\begin{array}{c}\vdots\,\Pi'\\ \Gamma' : e_y \Downarrow \Delta' : v_y\end{array}}{(\Gamma', y \mapsto e_y) : y \Downarrow (\Delta', y \mapsto v_y) : v_y}\ (\mathbf{var_y})$$

  where $(\Gamma', y \mapsto e_y) = \Gamma$ and $(\Delta', y \mapsto v_y) = \Delta$. If $x \in V(\Pi) \setminus \mathit{diff}(\Pi)$, then $x \in V(\Pi') \setminus \mathit{diff}(\Pi')$ too. By inductive hypothesis, $\Gamma'(x)$ is a value. Therefore, $\Gamma(x)$ is a value because $\Gamma(x) = \Gamma'(x)$. On the other hand, if $\Gamma(x)$ is a value, $\Gamma'(x)$ is also a value because $\Gamma(x) = \Gamma'(x)$. By inductive hypothesis, $x \in V(\Pi') \setminus \mathit{diff}(\Pi')$, i.e., $\Gamma'(x) = \Delta'(x)$. Thus, $\Gamma(x) = \Delta(x)$ (because $\Delta(x) = \Delta'(x)$) and the result follows from Definition 6.4.

- ($\mathbf{app}$). Π takes the form

$$\frac{\begin{array}{cc}\vdots\,\Pi_1 & \vdots\,\Pi_2\\ \Gamma : e \Downarrow \Theta : \lambda z.e' & \Theta : e'[y/z] \Downarrow \Delta : v\end{array}}{\Gamma : e\ y \Downarrow \Delta : v}\ (\mathbf{app}).$$

  Suppose that $x \in V(\Pi) \setminus \mathit{diff}(\Pi)$. There are two possibilities:

  - $x \in V(\Pi_1)$. We claim that $x \notin \mathit{diff}(\Pi)$ implies $x \notin \mathit{diff}(\Pi_1)$ and the result follows by inductive hypothesis. This claim can also be proved easily. Suppose that $x \in \mathit{diff}(\Pi_1)$. By Lemma 3.11, then, $\Theta(x)$ is a value v_x which, by Lemma 3.10, implies that $\Delta(x) = v_x$ too. This implies $x \in \mathit{diff}(\Pi)$ which is contradictory.
  - $x \notin V(\Pi_1)$. Therefore, $x \in V(\Pi_2)$, and, by Lemma 6.3, $\Gamma(x) = \Theta(x)$. Hence, $x \notin \mathit{diff}(\Pi_2)$ because otherwise $\Gamma(x) = \Theta(x) \neq \Delta(x)$, namely, $x \in \mathit{diff}(\Pi)$ which is contradictory. Thus, by inductive hypothesis, $\Theta(x)$ is a value. The result follows because $\Gamma(x) = \Theta(x)$.

  Suppose on the other hand that $x \in V(\Pi)$ and $\Gamma(x)$ is a value v. By Lemma 3.10, $\Delta(x) = v$. Accordingly, $\Gamma(x) = \Delta(x)$, i.e., $x \notin \mathit{diff}(\Pi)$.

- **(let)**. Π takes the form

$$\frac{\begin{array}{c}\vdots\,\Pi'\\ (\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow (\Delta, x_i \mapsto e_i')_{i=1}^n : v\end{array}}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : v}\ \textbf{(let)}.$$

  Suppose that $x \in V(\Pi) \setminus \mathit{diff}(\Pi)$. This implies that $x \in V(\Pi') \setminus \mathit{diff}(\Pi')$. By inductive hypothesis, then, $(\Gamma, x_i \mapsto e_i)_{i=1}^n$ is a value. The result follows by noting that $\Gamma(x) = (\Gamma, x_i \mapsto e_i)_{i=1}^n(x)$.

  Suppose on the other hand that $x \in V(\Pi)$ and $\Gamma(x)$ is a value. Therefore, $x \in V(\Pi')$ and $(\Gamma, x_i \mapsto e_i)_{i=1}^n(x)$ is a value (because $(\Gamma, x_i \mapsto e_i)_{i=1}^n(x) = \Gamma(x)$). By inductive hypothesis, $x \in V(\Pi') \setminus \mathit{diff}(\Pi')$, i.e., $(\Delta, x_i \mapsto e_i')_{i=1}^n(x) = (\Gamma, x_i \mapsto e_i)_{i=1}^n(x)$. Consequently, $\Gamma(x) = \Delta(x)$ and the result follows.

- **(seq)**. Π takes the form

$$\frac{\begin{array}{cc}\vdots\,\Pi_1 & \vdots\,\Pi_2\\ \Gamma : e_1 \Downarrow \Theta : v_1 & \Theta : e_2 \Downarrow \Delta : v_2\end{array}}{\Gamma : e_1\ \mathsf{seq}\ e_2 \Downarrow \Delta : v_2}\ \textbf{(seq)}.$$

  Suppose that $x \in V(\Pi) \setminus \mathit{diff}(\Pi)$. There are two possibilities:

  - $x \in V(\Pi_1)$. We claim that $x \notin \mathit{diff}(\Pi)$ implies $x \notin \mathit{diff}(\Pi_1)$ and the result follows by inductive hypothesis. This claim can also be proved easily. Suppose that $x \in \mathit{diff}(\Pi_1)$. By Lemma 3.11, then, $\Theta(x)$ is a value v_x which, by Lemma 3.10, implies that $\Delta(x) = v_x$ too. This implies $x \in \mathit{diff}(\Pi)$ which is contradictory.
  - $x \notin V(\Pi_1)$. Therefore, $x \in V(\Pi_2)$, and, by Lemma 6.3, $\Gamma(x) = \Theta(x)$. Hence, $x \notin \mathit{diff}(\Pi_2)$ because otherwise $\Gamma(x) = \Theta(x) \neq \Delta(x)$, namely, $x \in \mathit{diff}(\Pi)$ which is contradictory. Thus, by inductive hypothesis, $\Theta(x)$ is a value. The result follows because $\Gamma(x) = \Theta(x)$.

  Suppose on the other hand that $x \in V(\Pi)$ and $\Gamma(x)$ is a value v. By Lemma 3.10, $\Delta(x) = v$. Accordingly, $\Gamma(x) = \Delta(x)$, i.e., $x \notin \mathit{diff}(\Pi)$.

□

Corollary 6.6. *Suppose* $\Gamma : e_1 \Downarrow_{\Pi_1} \Delta_1 : v_1$ *and* $\Gamma : e_2 \Downarrow_{\Pi_2} \Delta_2 : v_2$ *such that* $V(\Pi_1) = V(\Pi_2)$. *Then,* $\mathit{diff}(\Pi_1) = \mathit{diff}(\Pi_2)$.

Proof. Suppose $\Gamma : e_1 \Downarrow_{\Pi_1} \Delta_1 : v_1$ and $\Gamma : e_2 \Downarrow_{\Pi_2} \Delta_2 : v_2$ such that $V(\Pi_1) = V(\Pi_2)$. Let $x \in \mathit{diff}(\Pi_1)$. If we can show that $x \in \mathit{diff}(\Pi_2)$, the result follows by symmetry. Suppose on the contrary that $x \notin \mathit{diff}(\Pi_2)$. Then, because $x \in V(\Pi_2)$, by Lemma 6.5, $\Gamma(x)$ is a value. On the other hand, because $x \in V(\Pi_1)$, by Lemma 6.5, this means that $x \notin \mathit{diff}(\Pi_1)$ — which is contradictory. □

Lemma 6.7. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$. *Then, for any* $x \in dom(\Gamma)$, *the following are equivalent:*

(1) $x \in \mathit{diff}(\Pi)$.

(2) $\Gamma(x)$ *is not a value and* $\Delta(x)$ *is a value.*

(3) Π *contains a non-trivial instance of* $(\mathbf{var_x})$.

Proof. Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$. We will prove that, for any $x \in dom(\Gamma)$, (1) $\Rightarrow$ (2) $\Rightarrow$ (3) $\Rightarrow$ (1).

(1) $\Rightarrow$ (2). Suppose $x \in \mathit{diff}(\Pi)$. By construction (Definition 6.4) this means that $\Gamma(x) \neq \Delta(x)$. Therefore, by Lemma 3.11, $\Delta(x)$ is a value. On the other hand, $\Gamma(x)$ cannot be a value, because otherwise, by Lemma 3.10, $\Gamma(x) = \Delta(x)$ which is contradictory.

(2) $\Rightarrow$ (3). Contrapositive of (the $\Rightarrow$ direction in) Lemma 6.3.

(3) $\Rightarrow$ (1). We prove the contrapositive. If $x \notin \mathit{diff}(\Pi)$, by construction (Definition 6.4), $\Gamma(x) = \Delta(x)$. By Lemma 6.3, therefore, Π contains no non-trivial instance of $(\mathbf{var_x})$, as desired.

□

Corollary 6.8. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$ *where* $x \in V(\Pi)$. *Then,* $\Delta(x)$ *is a value.*

Proof. Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$ where $x \in V(\Pi)$. There are two possibilities:

- If $x \in \mathit{diff}(\Pi)$, then, by Lemma 6.7, $\Delta(x)$ is a value.
- If $x \notin \mathit{diff}(\Pi)$, then, by Lemma 6.5, $\Delta(x)$ is a value.

□

We can now move on to some (slightly) less obvious results:

Theorem 6.9. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$. *For each* $x \in dom(\Gamma)$, Π *contains at most one non-trivial instance of* $(\mathbf{var_x})$.

Proof. Induction on Π based on the final rule in it:

- $(\mathbf{lam})$. Π takes the form

$$\frac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\ (\mathbf{lam}).$$

The result is correct because Π contains no $(\mathbf{var_x})$ instances.

- (**var$_x$**). Π takes the form

$$\frac{\begin{array}{c}\vdots\,\Pi'\\ \Gamma' : e_x \Downarrow \Delta' : v_x\end{array}}{(\Gamma', x \mapsto e_x) : x \Downarrow (\Delta', x \mapsto v_x) : v_x}\,(\textbf{var}_x).$$

  By construction $x \notin dom(\Gamma')$. Using our well-formedness assumption that let-bound variables are introduced distinct (Remark 3.5), it is easy to verify that $x \notin V(\Pi')$. The result follows.

- (**var$_y$**) for y other than x. Π takes the form

$$\frac{\begin{array}{c}\vdots\,\Pi'\\ \Gamma' : e_y \Downarrow \Delta' : v_y\end{array}}{(\Gamma', y \mapsto e_y) : y \Downarrow (\Delta', y \mapsto v_y) : v_y}\,(\textbf{var}_y)$$

  where $(\Gamma', y \mapsto e_y) = \Gamma$ and $(\Delta', y \mapsto v_y) = \Delta$. By inductive hypothesis, Π' contains at most once instance of (**var$_x$**). The result follows because Π only adds a (**var$_y$**) to Π'.

- (**app**). Π takes the form

$$\frac{\begin{array}{cc}\vdots\,\Pi_1 & \vdots\,\Pi_2\\ \Gamma : e \Downarrow \Theta : \lambda z.e' & \Theta : e'[y/z] \Downarrow \Delta : v\end{array}}{\Gamma : e\ y \Downarrow \Delta : v}\,(\textbf{app}).$$

  We now consider cases:

  - The case $x \notin \mathit{diff}(\Pi_1)$. By Lemma 6.7, Π_1 contains no non-trivial instance of (**var$_x$**). By inductive hypothesis, Π_2 contains at most one non-trivial instance of (**var$_x$**). The result follows.
  - The case $x \in \mathit{diff}(\Pi_1)$. By inductive hypothesis, Π_1 contains at most one non-trivial instance of (**var$_x$**). By Lemma 6.7, $\Theta(x)$ is a value. Furthermore, by Lemma 3.10, $\Delta(x)$ is a value too. By Lemma 6.7, again Π_2 contains no non-trivial instance of (**var$_x$**). The result follows.

- (**let**). Π takes the form

$$\frac{\begin{array}{c}\vdots\,\Pi'\\ (\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow (\Delta, x_i \mapsto e'_i)_{i=1}^n : v\end{array}}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : v}\,(\textbf{let}).$$

  By inductive hypothesis, Π' consists at most on instance of (**var$_x$**). The same is visibly the case for Π too because it only augments Π' by a (**let**).

- (**seq**). Π takes the form

$$\frac{\begin{array}{cc}\vdots\,\Pi_1 & \vdots\,\Pi_2\\ \Gamma : e_1 \Downarrow \Theta : v_1 & \Theta : e_2 \Downarrow \Delta : v_2\end{array}}{\Gamma : e_1 \;\mathsf{seq}\; e_2 \Downarrow \Delta : v_2}\,(\mathbf{seq}).$$

  There are two cases to consider:

  - The case $x \notin \mathit{diff}(\Pi_1)$. By Lemma 6.7, Π_1 contains no non-trivial instance of ($\mathbf{var_x}$). By inductive hypothesis, Π_2 contains at most one non-trivial instance of ($\mathbf{var_x}$). The result follows.
  - The case $x \in \mathit{diff}(\Pi_1)$. By inductive hypothesis, Π_1 contains at most one non-trivial instance of ($\mathbf{var_x}$). By Lemma 6.7, $\Theta(x)$ is a value. Furthermore, by Lemma 3.10, $\Delta(x)$ is a value too. By Lemma 6.7, again Π_2 contains no non-trivial instance of ($\mathbf{var_x}$). The result follows.

□

Lemma 6.10. *Suppose* $\Gamma : x \Downarrow_{\Pi_x} \Delta : v_x$ *and* $\mathit{diff}(\Pi_x) = \varnothing$. *Then,* $\Gamma(x) = v_x$.

Proof. By definition (Definition 6.4), $\mathit{diff}(\Pi_x) = \varnothing$ implies $\Gamma = \Delta$. On the other hand, by Lemma 5.3 we have $\Delta(x) = v_x$. □

6.2 Essential Parts of a Heap for a Derivation

The mission of this section is to provide criteria for distinguishing the essential parts of a heap for a derivation. That is, the parts of a heap Γ without which a derivation $\Gamma : e \Downarrow \Delta : v$ is not derivable. Theorem 6.23 formalises this.

Definition 6.11. *For an* $x \in \mathit{dom}(\Gamma)$, *define* $\Gamma[x \mapsto e]$ *by:*

$$\begin{array}{lll}\Gamma[x \mapsto e](x) = e & & \\ \Gamma[x \mapsto e](y) = \Gamma(y) & & y \in \mathit{dom}(\Gamma)\\ \Gamma[x \mapsto e](y) & \mathit{undefined} & \mathit{otherwise.}\end{array}$$

We extend this notation as

$$\begin{aligned}\Gamma[x_i \mapsto e_i]_{i=1}^{1} &= \Gamma[x_1 \mapsto e_1]\\ \Gamma[x_i \mapsto e_i]_{i=1}^{n} &= (\Gamma[x_i \mapsto e_i]_{i=1}^{n-1})[x_n \mapsto e_n].\end{aligned}$$

Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$ *and suppose* Π *contains no instance of* ($\mathbf{var_x}$). *Then for any* e' *define* $\Pi[x \mapsto e']$ *to be the labelled tree obtained from* Π *by replacing every heap* Θ *appearing in* Π *with* $\Theta[x \mapsto e']$ *if* $x \in \mathit{dom}(\Theta)$; *otherwise, we leave* Θ *unchanged.*

Lemma 6.12. *Suppose* $\Gamma(x) = e_x$. *Then,* $\Gamma[x \mapsto e_x] = \Gamma$.

Proof. By Definition 6.11,

$$\Gamma[x \mapsto e_x](x) = e_x = \Gamma(x).$$

□

Lemma 6.13. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$ *and* $\Gamma(x)$ *is a value* v_x*. Then,* $\Pi[x \mapsto v_x] = \Pi$*.*

Proof. By Lemma 3.10, for every heap Ξ in Π such that $x \in dom(\Xi)$, $\Xi(x) = v_x$. We use Lemma 6.12 to conclude the desired result. □

Lemma 6.14. *For any heap* Γ *and variable* y *such that* $y \notin dom(\Gamma)$

$$(\Gamma, y \mapsto e_y)[x \mapsto e_x] = (\Gamma[x \mapsto e_x], y \mapsto e_y).$$

Proof. Using Definition 3.3 and Definition 6.11, it is trivial to verify that:

$$(\Gamma, y \mapsto e_y)[x \mapsto e_x](y) = e_y = (\Gamma[x \mapsto e_x], y \mapsto e_y)(y)$$
$$(\Gamma, y \mapsto e_y)[x \mapsto e_x](x) = e_x = (\Gamma[x \mapsto e_x], y \mapsto e_y)(x)$$

and for any z other than x and y

$$(\Gamma, y \mapsto e_y)[x \mapsto e_x](z) = \Gamma(z) = (\Gamma[x \mapsto e_x], y \mapsto e_y)(z).$$

□

Lemma 6.15. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$*. Suppose* $x \in dom(\Pi)$ *but* $x \notin V(\Pi)$*. Then for any* e_x*,*

$$\Gamma[x \mapsto e_x] : e \Downarrow_{\Pi[x \mapsto e_x]} \Delta[x \mapsto e_x] : v$$
$$diff(\Pi[x \mapsto e_x]) = diff(\Pi)$$
$$V(\Pi[x \mapsto e_x]) = V(\Pi).$$

Proof of Lemma 6.15 comes in Appendix A.1.

Definition 6.16. *Suppose* Γ *is a heap and* S *is a set of variables. Write* $\Gamma|_S$ *for the function defined by:*

- $\Gamma|_S(x) = \Gamma(x)$ *if* $\Gamma(x)$ *is defined and* $x \in S$*.*
- $\Gamma|_S(x)$ *is undefined otherwise.*

We call this Γ ***restricted*** *to* S*.*

Symmetrically, write $\Gamma \setminus S$ *for the function defined by:*

- $(\Gamma \setminus S)(x) = \Gamma(x)$ *if* $\Gamma(x)$ *is defined and* $x \notin S$*.*
- $\Gamma \setminus S$ *is undefined otherwise.*

Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$ *and suppose* $V(\Pi) \subseteq S$*. Write* $\Pi|_S$ *for the labelled tree obtained by replacing every heap* Ξ *occurring in* Π*, with* $\Xi \setminus S'$ *where* $S' = dom(\Gamma) \setminus S$*. We call this* Π ***restricted*** *to* S*. Note that — because, by Lemma 3.7,* $dom(\Gamma) = dom(\Delta)$ *— there is no difference between* $\Gamma|_S$ *and* $\Gamma \setminus S'$ *or likewise between* $\Delta|_S$ *and* $\Delta \setminus S'$*. Having this said, we prefer to write* $\Gamma|_S : e \Downarrow_{\Pi|_S} \Delta|_S : v$*.*

Lemma 6.17. *Suppose that* $x \notin dom(\Gamma)$ *and* $\Gamma : e \Downarrow_\Pi \Delta : v$. *Then,* $(\Gamma, x \mapsto e_x) : e \Downarrow_{\Pi^+} (\Delta, x \mapsto e_x) : v$ *where*

$$\begin{aligned} \Pi &= \Pi^+|_{dom(\Gamma)} \\ V(\Pi^+) &= V(\Pi) \\ diff(\Pi^+) &= diff(\Pi). \end{aligned}$$

Proof of Lemma 6.17 comes in Appendix A.2.

Remark 6.18. In words, Π^+ in Lemma 6.17 is a copy of Π in which the binding $x \mapsto e_x$ is added to every heap. It is possible but tedious to formally describe this. As the only application is in Lemma 6.17, we omit a formal treatment however.

Corollary 6.19. *Suppose* $\Gamma \subseteq \Gamma'$ *and* $\Gamma : e \Downarrow_\Pi \Delta : v$. *Then, there exists a heap* Δ' *such that* $\Gamma' : e \Downarrow_{\Pi'} \Delta' : v$ *where*

$$\begin{aligned} \Delta &= \Delta'|_{dom(\Gamma)} \\ \Pi &= \Pi'|_{dom(\Gamma)} \\ V(\Pi') &= V(\Pi) \\ diff(\Pi') &= diff(\Pi). \end{aligned}$$

Proof. Let $\Gamma' \backslash \Gamma = \{x_i \mapsto e_i\}_{i=1}^n$. The result follows by a straightforward induction on n and using Lemma 6.17. □

Lemma 6.20. *Suppose* $(\Gamma^-, x \mapsto e_x) : e \Downarrow_\Pi (\Delta^-, x \mapsto e'_x) : v$ *and* $x \notin V(\Pi)$. *Then,* $\Gamma^- : e \Downarrow_{\Pi^-} \Delta^- : v$ *where*

$$\begin{aligned} \Pi^- &= \Pi|_{dom(\Gamma^-)} \\ V(\Pi^-) &= V(\Pi) \\ diff(\Pi^-) &= diff(\Pi). \end{aligned}$$

Proof. Similar to proof of Lemma 6.17. □

Corollary 6.21. *Suppose* $\Gamma' \subseteq \Gamma$ *and* $\Gamma : e \Downarrow_\Pi \Delta : v$ *such that* $dom(\Gamma \setminus \Gamma') \cap V(\Pi) = \varnothing$. *Then, there exists a heap* Δ' *such that* $\Gamma' : e \Downarrow_{\Pi'} \Delta' : v$ *where*

$$\begin{aligned} \Delta' &= \Delta|_{dom(\Gamma')} \\ \Pi' &= \Pi|_{dom(\Gamma')} \\ V(\Pi') &= V(\Pi) \\ diff(\Pi') &= diff(\Pi). \end{aligned}$$

Proof. Similar to proof of Corollary 6.19. □

Lemma 6.22. *The following statements are correct:*

- *If* $\Gamma : e \Downarrow_\Pi \Delta : v$ *and* $V(\Pi) \subseteq S$, *then* $\Gamma|_S : e \Downarrow_{\Pi|_S} \Delta|_S : v$.
- *If* $\Gamma|_S : e \Downarrow_{\Pi|_S} \Delta|_S : v$ *and* $\Gamma(y) = \Delta(y)$ *for every* $y \in dom(\Gamma) \setminus S$, *then* $\Gamma : e \Downarrow_\Pi \Delta : v$.

Proof. Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$ and $V(\Pi) \subseteq S$ for some S. By definition (Definition 6.16)

$$dom(\Gamma \setminus \Gamma|_S) \cap V(\Pi) \subseteq dom(\Gamma \setminus \Gamma|_S) \cap S = (dom(\Gamma) \setminus S) \cap S = \varnothing.$$

Thus, by Corollary 6.21, $\Gamma|_S : e \Downarrow_{\Pi|_S} \Delta|_S : v$.

Suppose on the other hand that $\Gamma|_S : e \Downarrow_{\Pi|_S} \Delta|_S : v$ such that $\Gamma(y) = \Delta(y)$ for every $y \in S' = dom(\Gamma) \setminus S$. By an induction on size of S' and using Lemma 6.17, we conclude $\Gamma : e \Downarrow_\Pi \Delta : v$, as expected. □

Theorem 6.23 expresses that if $\Gamma : e \Downarrow_\Pi \Delta : v$, then Π only depends on the Γ restricted to $V(\Pi)$:

Theorem 6.23. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$ *and suppose* $\Gamma(z) = \Gamma'(z)$ *for every* $z \in V(\Pi)$. *Then there exist* Δ' *and* Π' *such that:*

$$\begin{array}{ll} \Gamma' : e \Downarrow_{\Pi'} \Delta' : v & diff(\Pi') = diff(\Pi) \\ V(\Pi') = V(\Pi) & \Delta'(z) = \Delta(z) \text{ for every } z \in V(\Pi). \end{array}$$

Proof. Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$. Let $\hat{\Gamma} = \Gamma|_{V(\Pi)}$ and $\hat{\Delta} = \Delta|_{V(\Pi)}$. By Corollary 6.21, $\hat{\Gamma} : e \Downarrow_{\Pi|_{V(\Pi)}} \hat{\Delta} : v$ where

$$V(\Pi|_{V(\Pi)}) = V(\Pi) \tag{6.1}$$
$$diff(\Pi|_{V(\Pi)}) = diff(\Pi) \tag{6.2}$$

On the other hand, by Corollary 6.19, $\Gamma' : e \Downarrow_{\Pi'} \Delta' : v$ where

$$V(\Pi|_{V(\Pi)}) = V(\Pi') \tag{6.3}$$
$$diff(\Pi|_{V(\Pi)}) = diff(\Pi') \tag{6.4}$$

Hence, $V(\Pi) = V(\Pi')$ by (6.1) and (6.3), and $diff(\Pi) = diff(\Pi')$ by (6.2) and (6.4). The result follows by noting that $\Delta'|_{V(\Pi)} = \hat{\Delta} = \Delta|_{V(\Pi)}$. □

Chapter 7

Heaps and Atomic Variables

Now that we have identified trivial and non-trivial ($\mathbf{var_x}$) instances, and identified the essential parts of a heap for a derivation, we can move on to consider a notion of *unit of change*. Intuitively, this is computations which do cause changes in a heap, but cause the minimum change. For this, we particularly underpin the variables with such a property — which we call *atomic*. Definition 7.2 makes this formal, and the rest of this chapter explores the properties of atomic variables. But, first, we need some introduction:

Consider these heaps:

$$\Gamma_1 = (y \mapsto x,\ x \mapsto y)$$
$$\Gamma_2 = (y \mapsto x,\ x \mapsto \mathsf{let}\ z = \lambda z.z\ \mathsf{in}\ zz)$$
$$\Gamma_3 = (y \mapsto x,\ x \mapsto \lambda z.z)$$

These impose *evaluation dependencies* on variables: for Γ_1, to evaluate the expression bound to y we must evaluate that bound to x, and vice-versa; for Γ_2, to evaluate the expression bound to y we must evaluate that bound to x, but to evaluate the expression bound to x we need evaluate no other parts of the heap; for Γ_3, to evaluate the expression bound to y we do not need to evaluate the expression bound to x, because it is already evaluated. This leads us to the following definition:

Definition 7.1. *Suppose* Γ *is a heap. Define the* ***(evaluation) dependency order*** *by:*

$$x \preceq_\Gamma y \quad \textit{when} \quad \Gamma : y \Downarrow_{\Pi_y} \Delta_y : v_y \quad \textit{and} \quad x \in \textit{diff}(\Pi_y)$$

Suppose now that $\Gamma : e \Downarrow_\Pi \Delta : v$. It is a fact that in this case, $\preceq_\Gamma$ is acyclic on $\textit{diff}(\Pi)$. Intuitively this is because otherwise Π would 'go on forever' and therefore could not exist; consider for example trying to construct some Π of the form $\Gamma_1 : y \Downarrow_\Pi \Delta : v$.[1]

In this section we develop part of the theory of the dependency order. Definition 7.2 (atomic variables) characterises variables that are least, in the dependency order, and represent units of evaluation that update exactly one variable in

[1] A slight subtlety: $x \preceq_{\Gamma_2} y$, but as we have no theory of partial derivations, $x \not\preceq_{\Gamma_1} y$. As discussed, $x \not\preceq_{\Gamma_3} y$, because $\Gamma_3(x) = \lambda z.z$ is already evaluated.

the heap. Theorems 7.10 and 7.11 remove least elements from and add them to the initial heap of an evaluation; in effect, 'pruning' and 'restoring' least elements from $\preceq_\Gamma$. Finally, Theorem 7.12 establishes a useful and important part of the fundamental observation above — that for any Π, there must be *some* atomic variable in $\mathit{diff}(\Pi)$, whenever $\mathit{diff}(\Pi) \neq \varnothing$. This is all that we will need for the results to follow.

$\preceq_\Gamma$ is implicit in what follows but we do not need all of it. We just need that least elements always exist, and that the number of variables mentioned in a derivation is finite; therefore, we shall only need to talk about atomic variables. We leave a full treatment of the dependency order to future work.

Definition 7.2. *Call x **atomic** in Γ when there exist Δ_x, v_x, and Π_x such that $\Gamma : x \Downarrow_{\Pi_x} \Delta_x : v_x$ and $\mathit{diff}(\Pi_x) = \{x\}$. Write $\mathit{atomic}(\Gamma)$ for the set of atomic variable symbols in Γ.*

So x is atomic in Γ when it can be calculated without affecting the rest of the heap (note that if $\Gamma(x)$ is a value then x is not atomic in Γ; we only measure *non-trivial* evaluation in the evaluation dependency order). This notion, as it turns out, is extremely useful in the proofs to follow. Note that x is atomic with respect to a *heap* — not with respect to any particular evaluation $\Gamma : e \Downarrow_\Pi \Delta : v$.

We will prefer the following slightly more succinct characterisation of atomicity:

Lemma 7.3. *$x \in \mathit{atomic}(\Gamma)$ if and only if there exist v_x and Π_x such that $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ and $\mathit{diff}(\Pi_x) = \{x\}$.*

Proof. ($\Leftarrow$) Suppose $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ and $\mathit{diff}(\Pi_x) = \{x\}$. By Definition 7.2, taking $\Delta_x = \Gamma[x \mapsto v_x]$ suffices to show that x is atomic in Γ.

($\Rightarrow$) Suppose on the other hand that x is atomic in Γ. By Definition 7.2, there exists Δ_x, v_x, and Π_x such that $\Gamma : x \Downarrow_{\Pi_x} \Delta_x : v_x$ and $\mathit{diff}(\Pi_x) = \{x\}$. Because $\mathit{diff}(\Pi_x) = \{x\}$, by Definition 6.4, the only variable y for which $\Gamma(y) \neq \Delta(y)$ is x. Furthermore, by Lemma 5.3, $\Delta_x(x) = v_x$. Consequently, $\Delta_x = \Gamma[x \mapsto v_x]$, as required. □

Notation 7.4. *Hereafter, "_" will represent the wildcard in our notation, so that, by $\Gamma : e \Downarrow \Delta : \_$, we are clarifying our lack of interest in the final value obtained after evaluating e in Γ. Write $\Gamma : e \Downarrow_\Pi$ for $\Gamma : e \Downarrow_\Pi \_ : \_$.*

Definition 7.5 is related to Definition 6.11, as is the notation used, but the constructions have different preconditions and distinct properties.

Definition 7.5. *Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$ such that $x \in V(\Pi)$. Suppose also that $x \in \mathit{atomic}(\Gamma)$, so that in particular $\Gamma : x \Downarrow_{\Pi_x} \Delta_x : v_x$ for some Π_x, Δ_x, and v_x.*

We define a labelled tree $\Pi[x \mapsto v_x]$ by inductively transforming Π based on its final rule. For each case, when $x \notin \mathit{dom}(\Gamma)$, define $\Pi[x \mapsto v_x] = \Pi$. Cases when $x \in \mathit{dom}(\Gamma)$ are explained below:

- **(lam).** *Π takes the form*

$$\frac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\ \textbf{(lam)}.$$

Define $\Pi[x \mapsto v_x]$ *to be*

$$\frac{}{\Gamma[x \mapsto v_x] : \lambda x.e \Downarrow \Gamma[x \mapsto v_x] : \lambda x.e}\ (\mathbf{lam}).$$

- $(\mathbf{var_x})$. Π *takes the form*

$$\frac{\begin{array}{c}\vdots \\ \Gamma' : e_x \Downarrow \Gamma' : v_x\end{array}}{(\Gamma', x \mapsto e_x) : x \Downarrow (\Gamma', x \mapsto v_x) : v_x}\ (\mathbf{var_x}).$$

Define $\Pi[x \mapsto v_x]$ *to be*

$$\frac{\dfrac{}{\Gamma' : v_x \Downarrow \Gamma' : v_x}\ (\mathbf{lam})}{(\Gamma', x \mapsto v_x) : x \Downarrow (\Gamma', x \mapsto v_x) : v_x}\ (\mathbf{var_x}).$$

- $(\mathbf{let})$ *and* $(\mathbf{var_y})$ *for* y *other than* x. Π *takes the form*

$$\frac{\begin{array}{c}\vdots\ \Pi' \\ \Gamma' : e' \Downarrow \Delta' : v'\end{array}}{\Gamma : e \Downarrow \Delta : v}\ (\mathbf{r})$$

where $(\mathbf{r}) \in \{(\mathbf{var_y}), (\mathbf{let})\}$. *Define* $\Pi[x \mapsto v_x]$ *to be*

$$\frac{\begin{array}{c}\vdots\ \Pi'[x \mapsto v_x] \\ \Gamma'[x \mapsto v_x] : e' \Downarrow \Delta'[x \mapsto v_x] : v'\end{array}}{\Gamma[x \mapsto v_x] : e \Downarrow \Delta[x \mapsto v_x] : v}\ (\mathbf{r}).$$

- $(\mathbf{app})$ *and* $(\mathbf{seq})$. Π *takes the form*

$$\frac{\begin{array}{cc}\vdots\ \Pi_1 & \vdots\ \Pi_2 \\ \Gamma : e_1 \Downarrow \Theta : v_1 & \Theta : e_2 \Downarrow \Delta : v_2\end{array}}{\Gamma : e \Downarrow \Delta : v}\ (\mathbf{r})$$

where $(\mathbf{r}) \in \{(\mathbf{app}), (\mathbf{seq})\}$. *Define* $\Pi[x \mapsto v_x]$ *to be*

$$\frac{\begin{array}{cc}\vdots\ \Pi_1[x \mapsto v_x] & \vdots\ \Pi_2[x \mapsto v_x] \\ \Gamma[x \mapsto v_x] : e_1 \Downarrow \Theta[x \mapsto v_x] : v_1 & \Theta[x \mapsto v_x] : e_2 \Downarrow \Delta[x \mapsto v_x] : v_2\end{array}}{\Gamma[x \mapsto v_x] : e \Downarrow \Delta[x \mapsto v_x] : v}\ (\mathbf{r}).$$

Remark 7.6. The following points about the $(\mathbf{var_x})$ case in Definition 7.5 are worth considering:

- Recall from Theorem 6.9 that there will be at most one non-trivial instance of ($\mathbf{var_x}$) in Π. In fact, the only difference between Definition 7.5 and Definition 6.11 is that the former replaces the unique non-trivial instance of ($\mathbf{var_x}$) for an atomic x with a trivial one, whilst the latter does not.
- $\Gamma : x \Downarrow$ guarantees that $\Gamma' : x \Downarrow$ for every heap such that Π contains a sub-derivation $\Gamma' : x \Downarrow$. Furthermore, as we will see in Lemma 7.9, in the unique instance of ($\mathbf{var_x}$), x is atomic in such Γ' as well.
- Because $x \in \mathit{atomic}(\Gamma)$, by Lemma 7.3, for any instance of ($\mathbf{var_x}$) in Π like $(\Gamma', x \mapsto \_) : x \Downarrow (\Delta', x \mapsto \_) : v_x$, we are guaranteed that $\Gamma' = \Delta'$. Note that this includes both trivial and non-trivial instances of ($\mathbf{var_x}$).
- For any sub-derivation Π' of $\Gamma : e \Downarrow \Delta : v$ which contains no non-trivial instance of ($\mathbf{var_x}$), $\Pi'[x \mapsto v_x]$ in Definition 7.5 and $\Pi'[x \mapsto v_x]$ in Definition 6.11 coincide.

Lemmas 7.7 and 7.8 are key technical results, which are needed for Theorems 7.10 and 7.11:

Lemma 7.7. *Suppose $x \in \mathit{atomic}(\Gamma)$, so $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ for some Π_x and v_x, and $\mathit{diff}(\Pi_x) = \{x\}$.*

Suppose $\Gamma : y \Downarrow_{\Pi_y} \Delta_y : v$, for some y other than x. Then if $x \in V(\Pi_y)$ then $y \notin V(\Pi_x)$.

Proof. We prove the contrapositive. Suppose that $y \in V(\Pi_x)$. By assumption $\mathit{diff}(\Pi_x) = \{x\}$ and it follows by Lemma 6.5 that $\Gamma(y)$ is a value. Therefore, as demonstrated in the proof of Lemma 5.4, Π_y consists of an instance of ($\mathbf{lam}$) followed by an instance of ($\mathbf{var_y}$). Namely, $V(\Pi_y) = \{y\}$, and in particular, $x \notin V(\Pi_y)$. The result follows. □

Lemma 7.8. *Suppose $x \in \mathit{atomic}(\Gamma)$, so $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ for some Π_x and v_x, and $\mathit{diff}(\Pi_x) = \{x\}$.*

Suppose $\Gamma : e \Downarrow_{\Pi} \Delta : v$ and $x \notin V(\Pi)$.

Then $x \in \mathit{atomic}(\Delta)$, and moreover $\Delta : x \Downarrow \Delta[x \mapsto v_x] : v_x$.

Proof. Suppose $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ for some v_x and some Π_x such that $\mathit{diff}(\Pi_x) = \{x\}$. Suppose $\Gamma : e \Downarrow_{\Pi} \Delta : v$ and $x \notin V(\Pi)$.

By Lemma 6.5, $\Gamma(z)$ is a value for every $z \in V(\Pi_x) \setminus \{x\}$, and $\Delta(z) = \Gamma(z)$ by Lemma 3.10. By assumption $x \notin V(\Pi)$, hence by Lemma 6.3 $\Delta(x) = \Gamma(x)$. So,

$$\Gamma(z) = \Delta(z) \qquad \text{for every} \qquad z \in V(\Pi_x). \tag{7.1}$$

By Theorem 6.23, it follows that there exist Π'_x and Δ' such that $\Delta : x \Downarrow_{\Pi'_x} \Delta' : v_x$ and $\mathit{diff}(\Pi'_x) = \mathit{diff}(\Pi_x) = \{x\}$. Besides,

$$\Delta'(z) = \Gamma[x \mapsto v_x](z) \qquad \text{for every} \qquad z \in V(\Pi_x). \tag{7.2}$$

From (7.1) and (7.2), we conclude

$$\Delta(z) = \Delta'(z) \qquad \text{for every} \qquad z \in V(\Pi_x) \setminus \{x\} \tag{7.3}$$

$$\Delta'(x) = v_x. \tag{7.4}$$

On the other hand, by Lemma 6.3,

$$\Delta(z) = \Delta'(z) \qquad \text{for every} \qquad z \notin V(\Pi_x). \tag{7.5}$$

Therefore, $\Delta' = \Delta[x \mapsto v_x]$ as a result of (7.3), (7.4), and (7.5). □

Lemma 7.9. *Suppose* $\Gamma : x \Downarrow_{\Pi_x} \Delta : v_x$ *and suppose* $\Gamma(z) = \Gamma'(z)$ *for every* $z \in V(\Pi_x)$. *Then,* $x \in \mathit{atomic}(\Gamma)$ *implies that* $x \in \mathit{atomic}(\Gamma')$.

Proof. Suppose x is atomic in Γ. By Lemma 7.3, $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ where $\mathit{diff}(\Pi_x) = \{x\}$. By Theorem 6.23, therefore, there exists a heap Δ' such that $\Gamma' : x \Downarrow_{\Pi'_x} \Delta' : v_x$ and $\mathit{diff}(\Pi'_x) = \{x\}$. Consequently, x is atomic in Γ' by Lemma 7.2. The result follows. □

Theorem 7.10 shows how to reduce a computation in a heap Γ to a computation in a heap $\Gamma[x \mapsto v_x]$ which is in some sense 'simpler'. This simplification step will be used in key results to follow, including Theorem 8.4 and Lemma 9.1:

Theorem 7.10. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$ *and* $x \in V(\Pi)$.

Suppose $x \in \mathit{atomic}(\Gamma)$*; so in particular* $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ *for some* v_x. *Then* $\Delta(x) = v_x$ *and*

$$\Gamma[x \mapsto v_x] : e \Downarrow_{\Pi[x \mapsto v_x]} \Delta : v.$$

Furthermore,

$$\begin{aligned} \mathit{diff}(\Pi[x \mapsto v_x]) &= \mathit{diff}(\Pi) \setminus \{x\} \\ V(\Pi[x \mapsto v_x]) \cup V(\Pi_x) &= V(\Pi) \\ x &\in V(\Pi[x \mapsto v_x]). \end{aligned}$$

Proof of Theorem 7.10 comes in Appendix B.1.

Theorem 7.11. *Suppose* $x \in \mathit{atomic}(\Gamma)$*; so in particular* $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ *for some* v_x, *and* Π_x *such that* $\mathit{diff}(\Pi_x) = \{x\}$. *Suppose* $\Gamma[x \mapsto v_x] : e \Downarrow_{\Pi'} \Delta : v$ *and* $x \in V(\Pi')$. *Then* $\Gamma : e \Downarrow_\Pi \Delta : v$ *for a* Π *such that* $\Pi[x \mapsto v_x] = \Pi'$ *and* $x \in V(\Pi)$.

Proof of Theorem 7.11 comes in Appendix B.2.

Theorem 7.12. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$. *Then exactly one of the following possibilities holds:*

- $\mathit{diff}(\Pi) = \varnothing$.
- *There exists an* $x \in \mathit{diff}(\Pi)$ *such that* $x \in \mathit{atomic}(\Gamma)$.

Proof. By induction on Π based on the final rule:

- **(lam)**. Π takes the form

$$\frac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\ \textbf{(lam)}$$

in which case $\mathit{diff}(\Pi) = \varnothing$.

- **(var$_\mathbf{x}$)**. Π takes the form

$$\frac{\begin{array}{c}\vdots\ \Pi' \\ \Gamma' : e \Downarrow \Delta' : v\end{array}}{(\Gamma', x \mapsto e) : x \Downarrow (\Delta', x \mapsto v) : v}\ \textbf{(var}_\mathbf{x}\textbf{)}$$

where $(\Gamma', x \mapsto e) = \Gamma$ and $(\Delta', x \mapsto v) = \Delta$. If $\mathit{diff}(\Pi') \neq \varnothing$ then by inductive hypothesis there exists an $x' \in \mathit{diff}(\Pi')$ such that $x' \in \mathit{atomic}(\Gamma')$. By Lemma 7.9, $x' \in \mathit{atomic}(\Gamma)$, and the result follows.

Suppose $\mathit{diff}(\Pi') = \varnothing$. There are now two cases: If $\Gamma(x) \neq \Delta(x)$ then $x \in \mathit{atomic}(\Gamma)$. If $\Gamma(x) = \Delta(x)$ then $\mathit{diff}(\Pi) = \varnothing$. In either case, the result follows.

- **(app)**. Π takes the form

$$\frac{\begin{array}{cc}\vdots\ \Pi_1 & \vdots\ \Pi_2 \\ \Gamma : e \Downarrow \Theta : \lambda z.e' & \Theta : e'[y/z] \Downarrow \Delta : v\end{array}}{\Gamma : e\ y \Downarrow \Delta : v}\ \textbf{(app)}.$$

If $\mathit{diff}(\Pi_1) \neq \varnothing$, then, by inductive hypothesis, there exists a variable $x \in \mathit{dom}(\Gamma)$ such that $x \in \mathit{atomic}(\Gamma)$, as desired.

Suppose $\mathit{diff}(\Pi_1) = \varnothing$. If $\mathit{diff}(\Pi_2) = \varnothing$, then $\mathit{diff}(\Pi) = \varnothing$ too. Otherwise, note that $\mathit{diff}(\Pi_1) = \varnothing$ implies $\Gamma = \Theta$. Therefore, we can use the inductive hypothesis to conclude that there exists a variable $x \in \mathit{diff}(\Pi_2)$ such that $x \in \mathit{atomic}(\Theta)$. Hence, there exists $x \in \mathit{diff}(\Pi)$ such that $x \in \mathit{atomic}(\Gamma)$, as required.

- **(let)**. Π takes the form

$$\frac{\begin{array}{c}\vdots\ \Pi' \\ (\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow (\Delta, x_i \mapsto e_i')_{i=1}^n : v\end{array}}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : v}\ \textbf{(let)}.$$

If $\mathit{diff}(\Pi') = \varnothing$, then for every $y \in \mathit{dom}((\Gamma, x_i \mapsto e_i)_{i=1}^n)$, it should be the case that $(\Gamma, x_i \mapsto e_i)_{i=1}^n(y) = (\Delta, x_i \mapsto e_i')_{i=1}^n(y)$. Because x_is are fresh (Definition 3.4), for every $z \in \mathit{dom}(\Gamma)$, this implies $\Gamma(z) = \Delta(z)$. Hence, $\mathit{diff}(\Pi) = \varnothing$ too.

Suppose $\mathit{diff}(\Pi') \neq \varnothing$. Then, by inductive hypothesis, there exists a variable x which is atomic in $(\Gamma, x_i \mapsto e_i)_{i=1}^n$. If $x \in \mathit{dom}(\Gamma)$, then,

by Lemma 7.9, x is also atomic in Γ, as desired. If every such x is in $\{x_1, \ldots, x_n\}$, we claim that $\mathit{diff}(\Pi) = \varnothing$ and the result follows. Below we prove our claim:

Suppose on the contrary that $\mathit{atomic}((\Gamma, x_i \mapsto e_i)_{i=1}^n) \subseteq \{x_i\}_{i=1}^n$ and $\mathit{diff}(\Pi) \neq \varnothing$. Then, there exists a variable $y \in \mathit{diff}(\Pi)$ such that Π' contains a derivation $\Xi : y \Downarrow_{\Pi_y^+} \_ : v_y$ where $\Gamma \subseteq \Xi$. (See Remark 7.13.) By our freshness conditions (Definition 3.4), $V(\Pi_y^+) \subseteq \mathit{dom}(\Gamma)$. (See Remark 7.14.) Therefore, by Theorem 6.23, $\Gamma : y \Downarrow_{\Pi_y} \_ : v_y$ is derivable. By inductive hypothesis, there are two possibilities:

- $\mathit{diff}(\Pi_y) = \varnothing$. In this case, by Lemma 6.10, $\Gamma(y) = v_y$. Hence, by Lemma 3.10, $\Delta(x) = v_y$ as well which contradicts $y \in \mathit{diff}(\Pi)$.
- There exists a $z \in \mathit{diff}(\Pi_y)$ such that $z \in \mathit{atomic}(\Gamma)$. Then, because x_is are fresh, by Lemma 7.9, $z \in \mathit{atomic}((\Gamma, x_i \mapsto e_i)_{i=1}^n)$, which given that $z \in \mathit{dom}(\Gamma)$ contradicts $\mathit{atomic}((\Gamma, x_i \mapsto e_i)_{i=1}^n) \subseteq \{x_i\}_{i=1}^n$.

- (**seq**). Π takes the form

$$\dfrac{\begin{array}{cc}\vdots\,\Pi_1 & \vdots\,\Pi_2\\ \Gamma : e_1 \Downarrow \Theta : v_1 & \Theta : e_2 \Downarrow \Delta : v_2\end{array}}{\Gamma : e_1 \;\mathsf{seq}\; e_2 \Downarrow \Delta : v_2}\;(\mathbf{seq}).$$

If $\mathit{diff}(\Pi_1) \neq \varnothing$, then, by inductive hypothesis, there exists a variable $x \in \mathit{dom}(\Gamma)$ such that $x \in \mathit{atomic}(\Gamma)$, as desired.

Suppose $\mathit{diff}(\Pi_1) = \varnothing$. If $\mathit{diff}(\Pi_2) = \varnothing$, then $\mathit{diff}(\Pi) = \varnothing$ too. Otherwise, note that $\mathit{diff}(\Pi_1) = \varnothing$ implies $\Gamma = \Theta$. Therefore, we can use the inductive hypothesis to conclude that there exists a variable $x \in \mathit{diff}(\Pi_2)$ such that $x \in \mathit{atomic}(\Theta)$. Hence, there exists $x \in \mathit{diff}(\Pi)$ such that $x \in \mathit{atomic}(\Gamma)$, as required.

□

Remark 7.13. In words, y in the (**let**) case of Theorem 7.12 is the *first* variable in $\mathit{dom}(\Gamma)$ which gets evaluated in Π'. The graphical intuition is the bottom-most left-most x' for which ($\mathbf{var}_{\mathbf{x}'}$) occurs in Π'. Note that this obviously exists, or the contrary assumption $\mathit{diff}(\Pi) \neq \varnothing$ becomes false.

Remark 7.14. For the (**let**) case of Theorem 7.12, by construction (Definition 6.1), $x_i \in V(\Pi_y^+)$ implies $x_i \in \mathit{dom}(\Gamma)$ or $x_i \in \mathit{fv}(e')$ for some $\Gamma(x') = e'$. This contradicts the freshness side conditions of (**let**) in our operational semantics (Definition 3.4).

Chapter 8

Analogous Heaps

We can now begin to exploit the results of Section 7. First we define a notion of *analogous* heaps $\Gamma_1 \approx \Gamma_2$ (Definition 8.1). Think of an expression e as a transformation on heaps; if $\Gamma : e \Downarrow \Delta : v$ then e causes Γ to 'evolve' to Δ, and in doing so e calculates v. Then $\Gamma_1 \approx \Gamma_2$ when for any e, e calculates the same value v in both heaps, though the final heaps may differ (just as in life, when two situations are analogous then we expect the same actions to yield the same results).

In Theorem 8.4 we prove that as heaps evolve under evaluation, they remain analogous. Thus, heaps are different from a 'store', where the value associated to a variable may change arbitrarily. The proof-method is not induction on a derivation Π; instead, we exploit atomic elements and use induction on the size of $\mathit{diff}(\Pi)$ as we promised at the start of Section 7.

In fact, all that happens to the heap in evaluation is that a variable that was bound to an expression e, may become bound to a value — and because heaps remain analogous under evaluation, this value is the same as we would obtain by evaluating e directly in the intitial heap. This is Theorem 8.5, and Corollary 8.6 repackages part of that theorem.

One quite surprising consequence of all this, is that, in presence of a garbage-collecting (**let**) — like that of our operational semantics (Definition 3.4) — value equivalence $\cong_v$ implies heap equivalence $\cong_h$, and so also implies strict equivalence $\cong_s$ (Lemma 4.5 notes the reverse implication, which is essentially immediate). This is Lemma 8.10 and Theorem 8.11.

Finally, in Section 8.3, we formalise this alternate view about evaluations: heaps can be viewed as states of a state transition system which evolve to each other upon evaluation of expressions. Theorem 8.19 proves that $\approx$ is a bisimilarity relation for such a state transition system.

8.1 Evaluation of Heaps

We start by introducing a notion of equivalence *between heaps*, which — as shown in this chapter — will prove extremely useful:

Definition 8.1. *Define* $\Gamma_1 \approx \Gamma_2$ *by:*

$$\forall e. \forall v. \Big((\exists \Delta_1 . \Gamma_1 : e \Downarrow \Delta_1 : v) \Leftrightarrow (\exists \Delta_2 . \Gamma_2 : e \Downarrow \Delta_2 : v) \Big)$$

We call Γ_1 *and* Γ_2 ***analogous***.

Lemma 8.2. $\approx$ *is an equivalence relation (reflexive, transitive, symmetric).*

Proof. $\approx$ is reflexive because for any heap Γ:

$$\forall e, v.\ (\Gamma : e \Downarrow \Delta : v) \Leftrightarrow (\Gamma : e \Downarrow \Delta : v).$$

$\approx$ is symmetric because for any heaps Γ_1 and Γ_2,

$$\forall e, v.\ (\Gamma_1 : e \Downarrow \Delta : v) \Leftrightarrow (\Gamma_2 : e \Downarrow \Delta : v)$$

implies

$$\forall e, v.\ (\Gamma_2 : e \Downarrow \Delta : v) \Leftrightarrow (\Gamma_1 : e \Downarrow \Delta : v).$$

Finally, in order to show that $\approx$ is transitive, fix heaps Γ_1, Γ_2, and Γ_3 such that $\Gamma_1 \approx \Gamma_2$ and $\Gamma_2 \approx \Gamma_3$. By construction (Definition 8.1) then:

$$\forall e, v.\ (\Gamma_1 : e \Downarrow \Delta : v) \Leftrightarrow (\Gamma_2 : e \Downarrow \Delta : v) \tag{8.1}$$

$$\forall e', v'.\ (\Gamma_2 : e' \Downarrow \Delta : v') \Leftrightarrow (\Gamma_3 : e' \Downarrow \Delta : v') \tag{8.2}$$

Fix e_0 and v_0 such that $\Gamma_1 : e_0 \Downarrow \Delta : v_0$. By (8.1), it follows that $\Gamma_2 : e_0 \Downarrow \Delta : v_0$ which by (8.2) implies $\Gamma_3 : e_0 \Downarrow \Delta : v_0$. The result follows by symmetry. □

Lemma 8.3. *Suppose* $x \in \mathit{atomic}(\Gamma)$*, so in particular* $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ *for some* Π_x *and* v_x*.*
Then $\Gamma \approx \Gamma[x \mapsto v_x]$*.*

Proof. Suppose that $\Gamma : e \Downarrow_\Pi \Delta : v$. We are supposed to show that there exists a heap Δ' such that $\Gamma[x \mapsto v_x] : e \Downarrow \Delta' : v$. There are two cases:

- The case $x \notin V(\Pi)$. By Lemma 6.15, $\Gamma[x \mapsto v_x] : e \Downarrow \Delta[x \mapsto v_x] : v$. Take $\Delta' = \Delta[x \mapsto v_x]$.
- The case $x \in V(\Pi)$. By Theorem 7.10, $\Gamma[x \mapsto v_x] : e \Downarrow_{\Pi[x \mapsto v_x]} \Delta : v$. Take $\Delta' = \Delta$.

Suppose $\Gamma[x \mapsto v_x] : e \Downarrow_\Pi \Delta : v$. We should show that there exists a heap Δ' such that $\Gamma : e \Downarrow \Delta' : v$. There are two cases:

- The case $x \notin V(\Pi)$. Write $e_x = \Gamma(x)$. By Lemma 6.15, $\Gamma : e \Downarrow \Delta[x \mapsto e_x] : v$. Take $\Delta' = \Delta[x \mapsto e_x]$.
- The case $x \in V(\Pi)$. By Theorem 7.11, $\Gamma : e \Downarrow \Delta : v$. Take $\Delta' = \Delta$. □

Theorem 8.4. *If* $\Gamma : e \Downarrow_\Pi \Delta : v$ *then* $\Gamma \approx \Delta$*.*

Proof. By induction on the size of $\mathit{diff}(\Pi)$. There are two cases:

- The case $\mathit{diff}(\Pi) = \varnothing$. So $\Gamma = \Delta$. We use reflexivity of $\approx$ (Lemma 8.2).
- The case $\mathit{diff}(\Pi) \neq \varnothing$. By Theorem 7.12, there exists an $x \in \mathit{diff}(\Pi)$ such that $x \in \mathit{atomic}(\Gamma)$, so in particular $\Gamma : x \Downarrow \Gamma[x \mapsto v_x] : v_x$ for some v_x. By Lemma 8.3, $\Gamma \approx \Gamma[x \mapsto v_x]$. By Theorem 7.10, $\Gamma[x \mapsto v_x] : e \Downarrow_{\Pi[x \mapsto v_x]} \Delta : v$ (Definition 7.5). By Theorem 7.10, $\mathit{diff}(\Pi[x \mapsto v_x]) = \mathit{diff}(\Pi) \setminus \{x\}$. By inductive hypothesis, $\Gamma[x \mapsto v_x] \approx \Delta$. We use transitivity of $\approx$ (Lemma 8.2). □

One way to view Theorem 8.4 is as a *bisimulation* result; if we view expressions as actions causing transformations of heaps, then it is easy to use Definition 8.1 and Theorem 8.4 to show that analogous heaps can perform the same actions, and then evolve to analogous heaps. See Section 8.3.

Theorem 8.5. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$ *and suppose* $x \in \mathit{diff}(\Pi)$. *Then* $\Delta(x) = v_x$, *where* $\Gamma : x \Downarrow \Delta_x : v_x$ *for some* Δ_x.

Proof. Suppose $x \in \mathit{diff}(\Pi)$. By Lemma 6.7, $\Delta(x)$ is a value, write it v_x. It follows by Lemma 5.4 that $\Delta : x \Downarrow \Delta : v_x$. By Theorem 8.4, $\Gamma : x \Downarrow \Delta_x : v_x$ for some Δ_x. □

Note that Theorem 8.5 only augments Corollary 6.8 by clarifying the property of v_x that $\Gamma : x \Downarrow \Delta_x : v_x$ for some Δ_x. Theorem 8.4 is the result which makes this possible.

We say a little more about Theorem 8.5:

- If $\Gamma : e \Downarrow_\Pi \Delta : v$ then we can calculate Δ just from Γ and from knowing the set $V(\Pi)$; we do not have to know e or Π.
- Furthermore, we can do this calculation by evaluating each $x \in V(\Pi)$ starting from Γ; it is not necessary to evaluate the $x \in V(\Pi)$ in the order in which they are evaluated in Π (recall that **(app)** and **(seq)** make heaps 'evolve').

Corollary 8.6. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$ *and* $\Gamma : e' \Downarrow_{\Pi'} \Delta' : v'$. *If* $V(\Pi) = V(\Pi')$ *then* $\Delta = \Delta'$.

Proof. By Lemma 3.7, $\mathit{dom}(\Delta) = \mathit{dom}(\Gamma) = \mathit{dom}(\Delta')$. So, it suffices to prove $\Delta(x) = \Delta'(x)$ for every $x \in \mathit{dom}(\Gamma)$.

Suppose that $V(\Pi) = V(\Pi')$. By Corollary 6.6, $\mathit{diff}(\Pi) = \mathit{diff}(\Pi')$. Choose any $x \in V(\Pi)$. There are two cases:

- $x \in \mathit{diff}(\Pi) = \mathit{diff}(\Pi')$. By Theorem 8.5, $\Delta(x) = \Delta'(x) = v_x$ such that $\Gamma : x \Downarrow \Delta_x : v_x$ for some heap Δ_x.
- $x \in V(\Pi) \setminus \mathit{diff}(\Pi) = V(\Pi') \setminus \mathit{diff}(\Pi')$. By Lemma 6.5, it follows that $\Gamma(x)$ is a value v_x. By Lemma 3.10, then, $\Delta(x) = v_x$ and $\Delta'(x) = v_x$. □

8.2 Value Equivalence versus Strict Equivalence

So far, our results have used $\approx_s$ (we used $\approx_h$ once, in Theorem 4.6, and $\approx_v$ not at all). This is because $\approx_s$ establishes more properties than $\approx_v$ or $\approx_h$ (see Lemma 4.5). Conversely, $\approx_s$ is a harder proof-obligation than either $\approx_h$ or $\approx_v$. In this subsection we show that, unexpectedly, $\approx_v$ and $\approx_s$ are equal.

Definition 8.7. *Write Ω for the term* $\mathsf{let}\ z = \lambda x.(xx)\ \mathsf{in}\ zz$.

Lemma 8.8. *There exists no Δ and v such that $\Gamma : \Omega \Downarrow \Delta : v$.*

Proof. Suppose on the contrary that $\Gamma : \Omega \Downarrow_\Pi \Delta : v$ is derivable. Then, Π takes the form

$$\frac{\begin{array}{c}\vdots\ \Pi' \\ (\Gamma, z \mapsto \lambda x.xx) : zz \Downarrow (\Gamma, z \mapsto \lambda x.xx) : v\end{array}}{\Gamma : \Omega \Downarrow \Delta : v}\ (\textbf{let})$$

where Π' in return takes the form

$$\frac{\dfrac{}{\Gamma : \lambda x.xx \Downarrow \Gamma : \lambda x.xx}\ (\textbf{lam}) \qquad \begin{array}{c}\vdots\ \Pi'' \\ (\Gamma, z \mapsto \lambda x.xx) : zz \Downarrow (\Gamma, z \mapsto \lambda x.xx) : v\end{array}}{(\Gamma, z \mapsto \lambda x.xx) : zz \Downarrow (\Gamma, z \mapsto \lambda x.xx) : v}\ (\textbf{app}).$$

Therefore, by Determinism (Theorem 3.9), $\Pi' =_{\mathsf{let}\alpha} \Pi''$. (See Notation 3.8.) But, given that Π' contains Π'', this requires Π' to be infinite, which is contradictory. □

Lemma 8.9. *Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$ and $\Gamma(x) = \Omega$. Then $x \notin V(\Pi)$.*

Proof. Suppose $x \in \mathit{diff}(\Pi)$. By Theorem 8.5, there exist v_x, Δ_x, and Π_x such that $\Gamma : x \Downarrow_{\Pi_x} \Delta_x : v_x$ and $\Delta(x) = v_x$. By Lemma 8.8, it follows that $\Gamma(x) \neq \Omega$.

Now $\Gamma(x)$ is not a value, so by Lemma 6.5, $x \in V(\Pi) \setminus \mathit{diff}(\Pi)$ is impossible. The result follows. □

Lemma 8.10. *Assume $e_1 \approx_v e_2$ and suppose $\Gamma : e_1 \Downarrow_{\Pi_1} \Delta_1 : v$, so that also $\Gamma : e_2 \Downarrow_{\Pi_2} \Delta_2 : v$ for some Δ_2 and Π_1. Then $V(\Pi_1) = V(\Pi_2)$.*

As a corollary, $e_1 \approx_v e_2$ implies $e_1 \approx_h e_2$.

Proof. Take any $x \in V(\Pi_1) \setminus V(\Pi_2)$. We prove a contradiction.

By Lemma 6.15, $\Gamma[x \mapsto \Omega] : e_2 \Downarrow_{\Pi_2[x \mapsto \Omega]} \Delta_2[x \mapsto \Omega] : v$. It follows from $e_1 \approx_v e_2$ that $\Gamma[x \mapsto \Omega] : e_1 \Downarrow_{\Pi_1'} \Delta_1' : v$ for some Π_1' and Δ_1'. By Lemma 8.8, $x \notin V(\Pi_1')$. Write $e' = \Gamma(x)$. By Lemma 6.15, $\Gamma : e_1 \Downarrow_{\Pi_1'[x \mapsto e']} \Delta_1'[x \mapsto e'] : v$, and $V(\Pi_1'[x \mapsto e']) = V(\Pi_1')$. By Theorem 3.9, it follows that $\Pi_1'[x \mapsto e'] =_{\mathsf{let}\alpha} \Pi_1$. (See Notation 3.8.) Hence, by Lemma 6.2, $V(\Pi_1'[x \mapsto e']) = V(\Pi_1)$ — which contradicts our assumption that $x \in V(\Pi_1)$. The result follows by symmetry.

The corollary follows immediately, using Corollary 8.6. □

Theorem 8.11. *$e \approx_v e'$ if and only if $e \approx_s e'$.*

Proof. The right-to-left implication is Lemma 4.5. The left-to-right implication is by Lemmas 8.10 and 4.5. □

Remark 8.12. Note that $e \cong_h e'$ does not imply value equivalence $e \cong_v e'$. For example, from Corollary 9.2 it follows that $x \;\mathsf{seq}\; y \cong_h y \;\mathsf{seq}\; x$. However, it is a fact that $x \;\mathsf{seq}\; y \ncong_v y \;\mathsf{seq}\; x$.

Remark 8.13. It is noteworthy that Theorem 8.11 is only correct for Launchbury-based systems with garbage-collecting (**let**) rule. A large number of our results will fail without such a (**let**) rule. In particular, Lemma 8.10 will fail because Corollary 8.6 is not valid then. In a nutshell, the reason for this is that without a garbage-collecting (**let**) rule, expressiveness of heaps will grow upon evaluation of expressions. See Section 11.3 for further remarks.

8.3 Heap Bisimilarity

Although we have so far treated heaps as media for evaluation of expressions, an alternate view is also plausible. Heaps can be viewed as states of a state transition system which evolve to each other upon evaluation of expressions. Interestingly enough, $\approx$ is a bisimilarity relation for such a system. Lemma 8.17 and Theorem 8.19 formalise this.

We start from the definition of a bisimulation relation.

Definition 8.14. *Given a labelled state transition system $(S, \Lambda, \dot{\rightarrow})$, a **bisimulation** relation is a binary relation $\mathcal{R}$ over S when:*

For every $p, q \in S$ such that $p \;\mathcal{R}\; q$, and for all $\alpha \in \Lambda$

1. $\forall p' \in S.\ (p \xrightarrow{\alpha} p') \Rightarrow \exists q' \in S.\ (q \xrightarrow{\alpha} q') \wedge (p' \;\mathcal{R}\; q')$. *And,*
2. $\forall q' \in S.\ (q \xrightarrow{\alpha} q') \Rightarrow \exists p' \in S.\ (p \xrightarrow{\alpha} p') \wedge (p' \;\mathcal{R}\; q')$.

*Call the largest bisimulation relation a **bisimilarity**.*

Remark 8.15. The famous Tarski-Knaster Theorem guarantees the the existence of the largest bisimulation relation. See [79] and [32], for example, for proof and more.

Definition 8.16. *Let $\mathcal{H}$, $\mathcal{E}$, and $\mathcal{V}$ be the set of all heaps, expressions, and values, respectively. Define $\xrightarrow{(e,v)}$ for the transition system $\mathscr{T}_{\mathcal{H}} = (\mathcal{H}, \mathcal{E} \times \mathcal{V}, \xrightarrow{(.,.)})$ such that*

$$\Gamma_1 \xrightarrow{(e,v)} \Gamma_2 \text{ when } \Gamma_1 : e \Downarrow \Gamma_2 : v.$$

Lemma 8.17. *$\approx$ is a bimulation for $\mathscr{T}_{\mathcal{H}}$.*

Proof. Fix Γ_1 and Γ_2 for which $\Gamma_1 \approx \Gamma_2$ and $\Gamma_1 \xrightarrow{(e,v)} \Delta_1$ for some heap Δ_1. If we show that there exists a heap Δ_2 such that $\Gamma_2 \xrightarrow{(e,v)} \Delta_2$ and $\Delta_1 \approx \Delta_2$, the result follows by symmetry.

By construction (Definition 8.16), $\Gamma_1 \xrightarrow{(e,v)} \Delta_1$ means $\Gamma_1 : e \Downarrow_{\Pi_1} \Delta_1 : v$ for some Π_1. Given that $\Gamma_1 \approx \Gamma_2$, by construction (Definition 8.1), it follows that there exists a heap Δ_2 such that $\Gamma_2 : e \Downarrow \Delta_2 : v$. In order to show $\Delta_1 \approx \Delta_2$, suppose $\Delta_1 : e' \Downarrow_{\Pi_1'} \Delta_1' : v'$. We can use Π_1 and Π_1' to conclude $\Gamma_1 : e \;\mathsf{seq}\; e' \Downarrow \Delta_1' : v'$:

$$\frac{\begin{array}{cc}\vdots\,\Pi_1 & \vdots\,\Pi_1' \\ \Gamma_1 : e \Downarrow \Delta_1 : v & \Delta_1 : e' \Downarrow \Delta_1' : v'\end{array}}{\Gamma_1 : e \;\mathsf{seq}\; e' \Downarrow \Delta_1' : v'}\,(\mathbf{seq}).$$

Because $\Gamma_1 \approx \Gamma_2$, (by Definition 8.1), it follows that there exists a heap Δ_2' such that $\Gamma_2 : e \;\mathsf{seq}\; e' \Downarrow \Delta_2' : v'$ which takes the form

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma_2 : e \Downarrow \Xi : v & \Xi : e' \Downarrow \Delta_2' : v'\end{array}}{\Gamma_2 : e \;\mathsf{seq}\; e' \Downarrow \Delta_2' : v'}\,(\mathbf{seq}).$$

By Determinism (Theorem 3.9), $\Gamma_2 : e \Downarrow \Delta_2 : v$ and $\Gamma_2 : e \Downarrow \Xi : v$ imply $\Xi = \Delta_2$. Hence, $\Delta_2 : e' \Downarrow \Delta_2' : v'$, namely, $\Delta_1 \approx \Delta_2$ as desired. □

Remark 8.18. In the proof of Lemma 8.17, after establishing the existence of Δ_2, we could have concluded the desired result in the following way: by Theorem 8.4, $\Gamma_1 \approx \Delta_1$ and $\Gamma_2 \approx \Delta_2$. The result follows by transitivity of $\approx$ (Corollary 8.2).

Although this latter proof is correct and even shorter, we still preferred to use the former for Lemma 8.17. This is to show that Lemma 8.17 is independent of Theorem 8.4. Furthermore, the former proof emphasises the fact that seq can be viewed as **two** transition steps for $\mathscr{T}_{\mathcal{H}}$ (rather than a single one).

Theorem 8.19. *$\approx$ is a bisimilarity for $\mathscr{T}_{\mathcal{H}}$.*

Proof. Suppose $\mathcal{R}$ is a bisimulation for $\mathscr{T}_{\mathcal{H}}$. It is straightforward to prove $\mathcal{R} \subseteq \approx$, and the result is immediate: fix Γ_1 and Γ_2 such that $\Gamma_1 \mathcal{R} \Gamma_2$. Given that $\mathcal{R}$ is a bisimulation for $\mathscr{T}_{\mathcal{H}}$, by construction (Definition 8.16): $\Gamma_1 \xrightarrow{(e,v)} \Delta_1$ implies $\Gamma_2 \xrightarrow{(e,v)} \Delta_2$ for some heap Δ_2 (such that $\Delta_1 \mathcal{R} \Delta_2$). In other words, $\Gamma_1 : e \Downarrow \Delta_1 : v$ implies $\Gamma_2 : e \Downarrow \Delta_2 : v$ for some heap Δ_2 (such that $\Delta_1 \mathcal{R} \Delta_2$). By symmetry, it follows that $\Gamma_1 \approx \Gamma_2$. □

Chapter 9

Left-Commutativity

This chapter shows that the semantics of expressions can be preserved, even when the evaluation of certain subexpressions are reordered using selective strictness. The main result is the left-commutativity of seq (Theorem 9.3): that $(e_1 \ \mathsf{seq}\ e_2) \ \mathsf{seq}\ e_3 \cong_s (e_2 \ \mathsf{seq}\ e_1) \ \mathsf{seq}\ e_3$. By Theorem 4.6, it suffices to prove Corollary 9.2, that $e_1 \ \mathsf{seq}\ e_2 \cong_h e_2 \ \mathsf{seq}\ e_1$. Theorems 8.4 and 8.5 from Chapter 8 play a particularly useful role in the proofs of this chapter.

A key insight is a non-trivial compositionality result on the difference sets induced by $e_1 \ \mathsf{seq}\ e_2$ (Lemma 9.1). Suppose $\Gamma : e_1 \ \mathsf{seq}\ e_2 \Downarrow_\Pi \Delta : v$. So by the semantics for seq, $\Gamma : e_1 \Downarrow_{\Pi_1} \Theta : v_1$ and $\Theta : e_2 \Downarrow_{\Pi_2} \Delta : v$. In words, e_1 is evaluated in Γ and then e_2 is evaluated in Θ, a heap analogous to but not equal to the initial heap Γ. Now, $e_1 \ \mathsf{seq}\ e_2$ causes Γ to evolve by evaluating expressions associated with variables in $\mathit{diff}(\Pi)$, and this is equal to $\mathit{diff}(\Pi_1) \cup \mathit{diff}(\Pi_2)$. We will prove the non-trivial fact that $\mathit{diff}(\Pi)$ is equal to $\mathit{diff}(\Pi_1) \cup \mathit{diff}(\Pi_2')$, where $\Gamma : e_2 \Downarrow_{\Pi_2'} \Delta_2 : v_2$. The proof is not by induction on Π, but by induction on $\mathit{diff}(\Pi)$ using the results of Chapter 7.

Once we have the compositionality result, $e_1 \ \mathsf{seq}\ e_2 \cong_h e_2 \ \mathsf{seq}\ e_1$ (Corollary 9.2) follows by Theorem 8.5, and left-commutativity (Theorem 9.3) is immediate by Theorem 4.6.

Lemma 9.1. *Suppose* $\Gamma : e_1 \Downarrow_{\Pi_1} \Delta_1 : v_1$ *and* $\Gamma : e_2 \Downarrow_{\Pi_2} \Delta_2 : v_2$. *Then, for some* Π *and* Δ,

$$\begin{gathered} \Gamma : e_1 \ \mathsf{seq}\ e_2 \Downarrow_\Pi \Delta : v_2, \\ \mathit{diff}(\Pi) = \mathit{diff}(\Pi_1) \cup \mathit{diff}(\Pi_2), \quad \textit{and} \\ V(\Pi) = V(\Pi_1) \cup V(\Pi_2). \end{gathered}$$

Proof. We work by induction on the size of $\mathit{diff}(\Pi_1)$.

Suppose $\mathit{diff}(\Pi_1) = \varnothing$. So, $\Delta_1 = \Gamma$, and we construct Π as follows:

$$\frac{\begin{array}{cc} \vdots\,\Pi_1 & \vdots\,\Pi_2 \\ \Gamma : e_1 \Downarrow \Gamma : v_1 & \Gamma : e_2 \Downarrow \Delta_2 : v_2 \end{array}}{\Gamma : e_1 \ \mathsf{seq}\ e_2 \Downarrow \Delta_2 : v_2}\ (\mathbf{seq}).$$

Clearly, $\mathit{diff}(\Pi) = \mathit{diff}(\Pi_2)$, so $\mathit{diff}(\Pi) = \mathit{diff}(\Pi_1) \cup \mathit{diff}(\Pi_2)$. Also, $V(\Pi) = V(\Pi_1) \cup V(\Pi_2)$.

Suppose $\mathit{diff}(\Pi_1) \neq \varnothing$. By Theorem 7.12, there exists a $z \in \mathit{diff}(\Pi_1)$ such that $z \in \mathit{atomic}(\Gamma)$, so in particular $\Gamma : z \Downarrow_{\Pi_z} \Gamma[z \mapsto v_z] : v_z$ for some Π_z and v_z. By Theorem 7.10, $\Gamma[z \mapsto v_z] : e_1 \Downarrow_{\Pi_1[z \mapsto v_z]} \Delta_1 : v_1$ where

$$\mathit{diff}(\Pi_1[z \mapsto v_z]) = \mathit{diff}(\Pi_1) \setminus \{z\}, \tag{9.1}$$
$$V(\Pi_1[z \mapsto v_z]) \cup V(\Pi_z) = V(\Pi_1), \text{ and,} \tag{9.2}$$
$$z \in V(\Pi_1[z \mapsto v_z]). \tag{9.3}$$

There are two cases here:

- $z \in V(\Pi_2)$. Then, by Theorem 7.10, $\Gamma[z \mapsto v_z] : e_2 \Downarrow_{\Pi_2[z \mapsto v_z]} \Delta_2 : v_2$ where

$$\mathit{diff}(\Pi_2[z \mapsto v_z]) = \mathit{diff}(\Pi_2) \setminus \{z\}, \text{ and,} \tag{9.4}$$
$$V(\Pi_2[z \mapsto v_z]) \cup V(\Pi_z) = V(\Pi_2). \tag{9.5}$$

- $z \notin V(\Pi_2)$. Then, by Lemma 6.15,

$$\Gamma[z \mapsto v_z] : e_2 \Downarrow_{\Pi_2[z \mapsto v_z]} \Delta_2[z \mapsto v_z] : v_2$$

  ($\Pi_2[z \mapsto v_z]$ in the sense of Definition 6.11) where

$$\mathit{diff}(\Pi_2[z \mapsto v_z]) = \mathit{diff}(\Pi_2) = \mathit{diff}(\Pi_2) \setminus \{z\}, \text{ and,} \tag{9.6}$$
$$V(\Pi_2[z \mapsto v_z]) = V(\Pi_2). \tag{9.7}$$

In either case, by inductive hypothesis, there exist Π' and Δ' such that

$$\Gamma[z \mapsto v_z] : e_1 \ \mathsf{seq}\ e_2 \Downarrow_{\Pi'} \Delta' : v_2, \tag{9.8}$$
$$\mathit{diff}(\Pi') = \mathit{diff}(\Pi_1[z \mapsto v_z]) \cup \mathit{diff}(\Pi_2[z \mapsto v_z]), \text{ and,} \tag{9.9}$$
$$V(\Pi') = V(\Pi_1[z \mapsto v_z]) \cup V(\Pi_2[z \mapsto v_z]). \tag{9.10}$$

By substituting (9.1) and (9.4) (or (9.6)) into (9.9), we get

$$\mathit{diff}(\Pi') = \big(\mathit{diff}(\Pi_1) \cup \mathit{diff}(\Pi_2)\big) \setminus \{z\}. \tag{9.11}$$

Furthermore, (9.3) and (9.10) imply $z \in V(\Pi')$. So, by Theorem 7.11, $\Gamma : e_1$ seq $e_2 \Downarrow_{\Pi} \Delta' : v_2$ for some Π such that $\Pi[z \mapsto v_z] = \Pi'$. By Theorem 7.10,

$$\mathit{diff}(\Pi) = \mathit{diff}(\Pi') \cup \{z\}, \text{ and,} \tag{9.12}$$
$$V(\Pi') \cup V(\Pi_z) = V(\Pi). \tag{9.13}$$

Finally, by substituting (9.11) into (9.12), we get $\mathit{diff}(\Pi) = \mathit{diff}(\Pi_1) \cup \mathit{diff}(\Pi_2)$. Additionally, as required, $V(\Pi) = V(\Pi_1) \cup V(\Pi_2)$ follows from (9.13) and the fact that $z \in V(\Pi)$. □

Corollary 9.2. e_1 seq $e_2 \cong_h e_2$ seq e_1.

Proof. Suppose $\Gamma : e_1 \mathsf{seq}\ e_2 \Downarrow_\Pi \Delta : v_2$. We will show that $\Gamma : e_2 \mathsf{seq}\ e_1 \Downarrow \Delta : v_1$ for some v_1; the result will then follow by symmetry.

Π takes the form

$$\dfrac{\begin{array}{cc}\vdots\,\Pi_1 & \vdots\,\Pi_2\\ \Gamma : e_1 \Downarrow \Theta : v_1 & \Theta : e_2 \Downarrow \Delta : v_2\end{array}}{\Gamma : e_1 \mathsf{seq}\ e_2 \Downarrow \Delta : v_2}\ (\mathbf{seq}).$$

By Theorem 8.4, $\Gamma \approx \Theta$. Since $\Theta : e_2 \Downarrow \Delta : v_2$, $\Gamma : e_2 \Downarrow_{\Pi_2'} \Theta' : v_2$ for some Θ'. Also, by Theorem 8.4, $\Theta' \approx \Gamma$. Therefore, $\Theta' : e_1 \Downarrow_{\Pi_1'} \Delta' : v_1$ for some Δ'. We combine Π_2' and Π_1' to construct a derivation Π' of $\Gamma : e_2 \mathsf{seq}\ e_1 \Downarrow \Delta' : v_1$.

By Lemma 9.1, $V(\Pi) = V(\Pi') = V(\Pi_1) \cup V(\Pi_2)$ and $\mathit{diff}(\Pi) = \mathit{diff}(\Pi') = \mathit{diff}(\Pi_1) \cup \mathit{diff}(\Pi_2)$. We now reason by cases:

- Suppose $z \in \mathit{diff}(\Pi)$. By Theorem 8.5, $\Delta(z) = v_z = \Delta'(z)$, such that $\Gamma : z \Downarrow \_ : v_z$.
- Suppose $z \in \mathit{dom}(\Gamma) \setminus \mathit{diff}(\Pi)$. $\Delta(z) = \Gamma(z) = \Delta'(z)$ by Lemma 6.3.

The result follows. □

Theorem 9.3 (Left-commutativity)**.**
$(e_1 \mathsf{seq}\ e_2) \mathsf{seq}\ e_3 \cong_s (e_2 \mathsf{seq}\ e_1) \mathsf{seq}\ e_3$.

As a corollary, $e_1 \mathsf{seq}\ (e_2 \mathsf{seq}\ e_3) \cong_s e_2 \mathsf{seq}\ (e_1 \mathsf{seq}\ e_3)$.

Proof. From Corollary 9.2 and Theorem 4.6.

The corollary follows by associativity (Theorem 5.1). □

Chapter 10

Idempotence

This chapter proves a general form of idempotence: $e \;\mathsf{seq}\; e \cong_s e$, extending the more restricted Theorem 5.5 ($x \;\mathsf{seq}\; x \cong_s x$). Given the results we have proved since then, the general version is not hard. First, we need a very natural property: once an expression has been evaluated, evaluating it again in the resulting heap has no further effect on that heap. This is Theorem 10.1. This chapter borrows heavily from [71].

Theorem 10.1. *If* $\Gamma : e \Downarrow \Delta : v$ *then* $\Delta : e \Downarrow \Delta : v$.

Proof. Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$. We work by induction on the size of $\mathit{diff}(\Pi)$. There are two cases:

- The case $\mathit{diff}(\Pi) = \varnothing$. Then, by definition (Definition 6.4), $\Gamma = \Delta$ and we are done.
- The case $\mathit{diff}(\Pi) \neq \varnothing$. By Theorem 7.12, there exists an $x \in \mathit{diff}(\Pi)$ such that $x \in \mathit{atomic}(\Gamma)$, so in particular $\Gamma : x \Downarrow \Gamma[x \mapsto v_x] : v_x$ for some v_x. By Lemma 6.7 $x \in V(\Pi)$, so by Theorem 7.10, $\Gamma[x \mapsto v_x] : e \Downarrow_{\Pi[x \mapsto v_x]} \Delta : v$ and $\mathit{diff}(\Pi[x \mapsto v_x]) = \mathit{diff}(\Pi) \setminus \{x\}$. (See Definition 7.5.) By inductive hypothesis, $\Delta : e \Downarrow \Delta : v$. □

The result follows.

Theorem 10.2 (Idempotence)**.** $e \;\mathsf{seq}\; e \cong_s e$.

Proof. Suppose that $\Gamma : e \;\mathsf{seq}\; e \Downarrow_\Pi \Delta : v$. Then Π must have the form:

$$\frac{\begin{array}{cc}\vdots\,\Pi_1 & \vdots\,\Pi_2\\ \Gamma : e \Downarrow \Theta : v & \Theta : e \Downarrow \Delta : v\end{array}}{\Gamma : e \;\mathsf{seq}\; e \Downarrow \Delta : v}\;(\mathbf{seq}).$$

By Theorem 10.1, $\Theta : e \Downarrow \Theta : v$. By Theorem 3.9, $\Theta = \Delta$. The result follows. □

Finally, we exploit associativity and idempotence (Theorems 5.1 and 10.2) to obtain a nice corollary:

Corollary 10.3. *The following equivalences are correct:*

1. $e_1 \;\mathsf{seq}\; (e_1 \;\mathsf{seq}\; e_2) \cong_s e_1 \;\mathsf{seq}\; e_2$
2. $(e_1 \;\mathsf{seq}\; e_2) \;\mathsf{seq}\; e_2 \cong_s e_1 \;\mathsf{seq}\; e_2$

Proof. By associativity (Theorem 5.1) it suffices to prove

$$(e_1 \;\mathsf{seq}\; e_1) \;\mathsf{seq}\; e_2 \cong_s e_1 \;\mathsf{seq}\; e_2 \quad \text{and}$$
$$e_1 \;\mathsf{seq}\; (e_2 \;\mathsf{seq}\; e_2) \cong_s e_1 \;\mathsf{seq}\; e_2.$$

This follows by idempotence (Theorem 10.2) and Theorem 4.7. □

Chapter 11

Conclusion and Future Work

In this chapter, we first present an overall summary of the research in this thesis (Section 11.1). We next provide a big-picture explanation about our proof techniques in Section 11.2. Finally, in Section 11.3, we end this chapter with a discussion on the limitations of this thesis, as well as possible directions for future work.

11.1 Summary

Selective strictness is a widely used, but potentially dangerous technique in call-by-need languages. The equivalences we prove in this thesis are new, as are the proof-methods.

This thesis begins by reviewing the related work (Chapter 2). We then start our technical development by considering three operational notions of program equivalence (Chapter 3). Each equivalence is based on observing a different part of the program's behaviour: the value returned ($\approx_v$), the final heap ($\approx_h$), or both ($\approx_s$) (Chapter 4). Subsequent chapters use the strongest equivalence $\approx_s$, but as it turns out, because of the garbage-collecting **(let)** rule, $\approx_v$ coincides with $\approx_s$ (Section 8.2).

In Chapter 5, we make two developments: We first prove Associativity of seq (Theorem 5.1) and Idempotence of seq for variables (Theorem 5.5). Next, we discuss a subtlety of the operational semantics of seq (Section 5.2).

Proving the Left-Commutativity of seq is far more challenging, as it entails proving equivalences between expressions where the order of evaluation of subexpressions has been changed in arbitrary ways. Chapters 6, 7, and 8 provide interesting insights while assembling the necessary machinery.

We find it important to distinguish between *non-trivial* uses of a variable that cause evaluation, and hence change the heap, and those *trivial* uses that do not. We identify the *difference* set of non-trivial variables in a derivation Π as $\mathit{diff}(\Pi)$. Finally, we show that expressions are indeed evaluated at most once (Chapter 6).

We develop some theory of a dependency relation between variables, imposed by heaps. We identify a particular special case of *atomic variables*, which are least in the dependency order, and which represent units of evaluation that update

exactly one variable in the heap. Theorem 7.12 is a key result showing that for any Π there must be *some* atomic variable in $\mathit{diff}(\Pi)$ (Section 7).

By viewing expressions as heap transformers, we define a notion of *analogous* heaps, and show that as heaps evolve under evaluation, they remain analogous (Chapter 8). This gives rise to a notion of *bisimilarity* between heaps (Section 8.3).

The results from Chapters 6 to 8 are applied to show that the the call-by-need semantics of expressions can be preserved, even when the evaluation of subexpressions are reordered using selective strictness. The main results are Left-Commutativity and Idempotence of seq (Theorems 9.3 and 10.2, respectively). Key insights are a non-trivial compositionality result on the difference sets, the commutativity of seq under heap equivalence (Chapter 9), and idempotence for expressions considered as heap-transformers (Theorem 10.1).

11.2 Proof Techniques

We now comment on aspects of our proof-techniques:

Because evaluation of subexpressions may be reordered in particularly complex ways by selective strictness, straightforward inductive principles seem to fail; instead we develop the basic properties of $\mathit{diff}(\Pi)$ (Chapter 6) and then in three central technical results, Theorems 7.10, 7.11, and 7.12, we lay the groundwork for simple and compact inductions found in Theorem 8.4, Lemma 9.1, and Theorem 10.1. These, in turn, make possible the proofs of our main results in Chapters 8, 9 and 10.

In the course of these proofs, an interesting and subtle interplay becomes apparent between heaps and derivations: given a heap Γ and an expression e, we may be able to form a derivation Π for evaluating e in Γ (which by Theorem 3.9 is unique up to fresh naming, if it exists). In Π we can observe where the expression bound to a given variable x is updated (if at all); this occurs at the unique (by Theorem 6.9) non-trivial instance of $(\mathbf{var_x})$. However, it is a fact that this order of update must be a refinement of the dependency order imposed by the heap Γ. Thus, in some sense, information flows from a heap Γ to the structure of any derivations built using it. On the other hand, it is not *a priori* obvious that the dependency order provides any useful information, as the example of $\Gamma_1 = (x \mapsto y,\ y \mapsto x)$ in Chapter 7 showed; no derivation can exist using x and y in this heap. However, Theorem 7.12 guarantees that if a derivation Π *does* exist, then Γ must provide at least one atomic element (least in the dependency order). In this sense, information also flows back from derivations to heaps. We can see this in our technical results; for example in Theorems 7.10 and 7.11 information flows from the heap to derivations, whereas in Theorem 7.12 we start from the derivation and deduce properties of the heap. The influence of this interplay extends beyond the proofs, to our most basic definitions. Thus, we do *not* define x to be atomic with respect to a derivation Π; we do not write in Definition 7.2 that x is 'atomic' when $\mathit{diff}(\Pi_x) = \{x\}$ for Π_x the subderivation of Π above a non-trivial instance of $(\mathbf{var_x})$ in Π. If we did, then proofs would fail,

because transformations of Π may change the order of evaluation and enlarge the subderivation such that $\mathit{diff}(\Pi_x)$ becomes non-empty.

11.3 Limitations and Future Work

As exemplified in Remark 3.5, the side condition $x_i \notin \mathit{fv}(v)$ for our garbage-collecting (**let**) rule causes arguably reasonable lazy programs to fail to reduce in our system. Here is an example of the situations which motivated us for having this side condition. Let us suppose, for the sake of argument, that we had removed the $x_i \notin \mathit{fv}(v)$ side condition from our (**let**) rule. Suppose also the following

$$\begin{aligned} e'' &= \mathsf{let}\ x_2 = \lambda x.x\ \mathsf{in}\ \lambda x.x_2, \\ e' &= e''\ x_1, \\ e &= \mathsf{let}\ x_1 = \lambda x.x\ \mathsf{in}\ e', \\ \Gamma_1 &= \{x_1 \mapsto \lambda x.x\},\ \text{and}, \\ \Gamma_{12} &= \Gamma_1 \cup \{x_2 \mapsto \lambda x.x\}. \end{aligned}$$

Then, $\varnothing : e \not\Downarrow$. Suppose otherwise, namely, suppose that $\varnothing : e \Downarrow$. We illustrate the problem by considering the derivation. If $\varnothing : e \Downarrow$ was derivable in our system, it would have taken the form

$$\dfrac{\dfrac{\dfrac{}{\Gamma_{12} : \lambda x.x_2 \Downarrow \Gamma_{12} : \lambda x.x_2}\,(\textbf{lam})}{\Gamma_1 : e'' \Downarrow \Gamma_1 : \lambda x.x_2}\,(\textbf{let})^* \qquad \begin{array}{c}\vdots\,\Pi\\ \Gamma_1 : x_2 \Downarrow \_ : \_\end{array}}{\Gamma_1 : e' \Downarrow \_ : \_}\,(\textbf{app})}{\varnothing : e \Downarrow \_ : \_}\,(\textbf{let}).$$

(Note that (**let**)* is not a valid instance of (**let**) because $x_2 \in \mathit{fv}(\lambda x.x_2)$.) However, Π cannot exist because x_2 is not bound in Γ_1, and therefore, there is no rule to apply. In fact, although x_2 was bound in Γ_{12}, the (erroneous) garbage-collecting (**let**)* rule takes it back. This is whilst x_2 is still hiding behind a λ-abstraction (in $\lambda x.x_2$).

It is natural to question the necessity of garbage-collection here. This question becomes more appropriate when one realises that real-world Haskell programmers might use programs such as e or e'' in practice. Interestingly enough, for non-lazy programming languages, it is arguable that programs like e'' must be banned because they are letting x_2 out of its scope. However, this is not a completely reasonable argument in presence of lazy evaluation. The reason is that, in lazy evaluation, we never know when bindings are going to be used. Therefore, whilst scope-violation errors should be caught using other syntactic scrutinies, the let-bindings should remain in the heap until they are finally used, if ever.

Whilst it might be possible to defend programs like e or e'' this way, there is no reason why an operational semantics should allow a program like $e''' = (\mathsf{let}\ x = \lambda y.y\ \mathsf{in}\ \lambda z.z)\ \mathsf{seq}\ x$. Unfortunately, any non-garbage-collecting (**let**) rule would cause programs such as e''' to reduce successfully. The reason is that, as also mentioned in Remark 8.13, the expressiveness of heaps will unreasonably grow

without garbage-collection. For example, the binding $x \mapsto \lambda y.y$ will unreasonably remain in the heap when the evaluation of $\mathsf{let}\ x = \lambda y.y\ \mathsf{in}\ \lambda z.z$ is complete. Thus, the operational semantics has no way to stop evaluation of x afterwards whilst x has no meaning from that point on (namely, after the evaluation of $\mathsf{let}\ x = \lambda y.y\ \mathsf{in}\ \lambda z.z$).

One might argue that catching errors such as the one in e''' is not a task for the operational semantics. Although this is partly correct, one needs to note that if the operational semantics fails to catch such errors, then pointless equivalences such as $e''' \cong_v \lambda z.z$ will arise. In other words, e''' would be a valid citizen of language as far as the operational semantics and observational equivalence are concerned. This can well foster numerous mysterious implementation bugs for code optimisers which typically work in the total trust of the soundness of observational equivalence(s) provided.

Several possibilities are worth studying for retaining garbage-collection whilst avoiding the problems mentioned above. Dr. Gabbay suggested extending the definition of values so that it also contains let-bindings of the form $\mathsf{let}\ \{x_i = e_i\}_{i=1}^n\ \mathsf{in}\ \lambda x.e$. This would be similar to the syntactic category "Answers" in [10]. On the other hand, the student suggested a pre-normalisation of the form:

$$(\mathsf{let}\ \{x_i = e_i\}_{i=1}^n\ \mathsf{in}\ \lambda x.e)\ y \rightsquigarrow \mathsf{let}\ \{x_i = e_i\}_{i=1}^n\ \mathsf{in}\ ((\lambda x.e)\ y).$$

(Note that this is not new to the literature. Ariola et al. [10, 7, 62] all have similar rules. See rules (let-C), and (C) rules in Figures 2.6 and 2.8, respectively, as well as $lift_{let}$ in [7].) Both these suggestion try to put every application *into its context* whilst they retain garbage-collection. Whether or not any of these possibilities, or a combination of them, will fix the problem is a subject for future work.

As another possibility for future work, one can investigate how to extend our proof-techniques to reason about selective strictness in conjunction with parallel evaluation. This is motivated by languages like GpH [103] and Eden [60] that combine selective strictness and parallelism. For example GpH uses both sequential and parallel compositions, seq and par, where par evaluates both arguments in parallel.

Appendix A

Additional Semantic Property Proofs

This chapter contains the proofs we postponed in Chapter 6. This includes Lemma 6.15 and Lemma 6.17, which are presented in A.1 and A.2, respectivey.

A.1 Lemma 6.15

Lemma 6.15. Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$. Suppose $x \in dom(\Pi)$ but $x \notin V(\Pi)$. Then for any e_x,

$$\Gamma[x \mapsto e_x] : e \Downarrow_{\Pi[x \mapsto e_x]} \Delta[x \mapsto e_x] : v$$
$$diff(\Pi[x \mapsto e_x]) = diff(\Pi)$$
$$V(\Pi[x \mapsto e_x]) = V(\Pi).$$

Proof. We inductively transform Π into $\Pi[x \mapsto e_x]$. We reason by cases on the final rule in Π:

- (**lam**). Π takes the form

$$\frac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\ (\textbf{lam})$$

and $\Pi[x \mapsto e_x]$ takes the form

$$\frac{}{\Gamma[x \mapsto e_x] : \lambda x.e \Downarrow \Gamma[x \mapsto e_x] : \lambda x.e}\ (\textbf{lam}).$$

The result is straightforward because $V(\Pi) = V(\Pi[x \mapsto e_x]) = \varnothing$ and $diff(\Pi) = diff(\Pi[x \mapsto e_x]) = \varnothing$.

- ($\textbf{var}_\textbf{y}$) for y other than x. Π takes the form

$$\frac{\begin{array}{c}\vdots\ \Pi' \\ \Gamma' : e_y \Downarrow \Delta' : v_y\end{array}}{(\Gamma', y \mapsto e_y) : y \Downarrow (\Delta', y \mapsto v_y) : v_y}\ (\textbf{var}_\textbf{y})$$

where $(\Gamma', y \mapsto e_y) = \Gamma$ and $(\Delta', y \mapsto v_y) = \Delta$. Given that $dom(\Gamma') = dom(\Gamma) \setminus \{y\}$, $x \in dom(\Gamma)$ implies $x \in dom(\Gamma')$. So, by inductive hypothesis

$$\begin{gathered}\Gamma'[x \mapsto e_x] : e_y \Downarrow_{\Pi'[x \mapsto e_x]} \Delta'[x \mapsto e_x] : v_y \\ \mathit{diff}(\Pi'[x \mapsto e_x]) = \mathit{diff}(\Pi') \\ V(\Pi'[x \mapsto e_x]) = V(\Pi').\end{gathered}$$

Given that $y \notin dom(\Gamma')$, by Definition 6.1, $V(\Pi') = V(\Pi) \setminus \{y\}$, in particular, $x \notin V(\Pi')$. Similarly, by Definition 6.4, $\mathit{diff}(\Pi') = \mathit{diff}(\Pi) \setminus \{y\}$ and

$$\frac{\begin{array}{c}\vdots\ \Pi'[x \mapsto e_x] \\ \Gamma'[x \mapsto e_x] : e_y \Downarrow \Delta'[x \mapsto e_x] : v_y\end{array}}{(\Gamma'[x \mapsto e_x], y \mapsto e_y) : y \Downarrow (\Delta'[x \mapsto e_x], y \mapsto v_y) : v_y}\ (\mathbf{var_y}).$$

The result follows by noting that, by Lemma 6.14, $(\Gamma'[x \mapsto e_x], y \mapsto e_y) = (\Gamma', y \mapsto e_y)[x \mapsto e_x]$ and $(\Delta'[x \mapsto e_x], y \mapsto v_y) = (\Delta', y \mapsto v_y)[x \mapsto e_x]$.

- $(\mathbf{var_x})$. This case is out of consideration because $x \notin V(\Pi)$.
- $(\mathbf{app})$. Π takes the form

$$\frac{\begin{array}{cc}\vdots\ \Pi_1 & \vdots\ \Pi_2 \\ \Gamma : e \Downarrow \Theta : \lambda z.e' & \Theta : e'[y/z] \Downarrow \Delta : v\end{array}}{\Gamma : e\ y \Downarrow \Delta : v}\ (\mathbf{app}).$$

$x \notin V(\Pi)$ implies $x \notin V(\Pi_1)$ and $x \notin V(\Pi_2)$. On the other hand, by Lemma 3.7, $x \in dom(\Gamma)$ implies $x \in dom(\Theta)$. So, by inductive hypothesis, $\Gamma[x \mapsto e_x] : e \Downarrow_{\Pi_1[x \mapsto e_x]} \Theta[x \mapsto e_x] : \lambda z.e'$ and $\Theta[x \mapsto e_x] : e'[y/z] \Downarrow_{\Pi_2[x \mapsto e_x]} \Delta[x \mapsto e_x] : v$. Thus

$$\frac{\begin{array}{cc}\vdots\ \Pi_1[x \mapsto e_x] & \vdots\ \Pi_2[x \mapsto e_x] \\ \Gamma[x \mapsto e_x] : e \Downarrow \Theta[x \mapsto e_x] : \lambda z.e' & \Theta[x \mapsto e_x] : e'[y/z] \Downarrow \Delta[x \mapsto e_x] : v\end{array}}{\Gamma[x \mapsto e_x] : e\ y \Downarrow \Delta[x \mapsto e_x] : v}\ (\mathbf{app}).$$

Clearly, this derivation is $\Pi[x \mapsto e_x]$. By inductive hypothesis:

$$\begin{gathered}\mathit{diff}(\Pi_1[x \mapsto e_x]) = \mathit{diff}(\Pi_1) \\ V(\Pi_1[x \mapsto e_x]) = V(\Pi_1)\end{gathered}$$

and

$$\begin{gathered}\mathit{diff}(\Pi_2[x \mapsto e_x]) = \mathit{diff}(\Pi_2) \\ V(\Pi_2[x \mapsto e_x]) = V(\Pi_2).\end{gathered}$$

The result follows by noting that

$$\begin{gathered}V(\Pi[x \mapsto e_x]) = V(\Pi_1[x \mapsto e_x]) \cup V(\Pi_2[x \mapsto e_x]) \\ \mathit{diff}(\Pi[x \mapsto e_x]) = \mathit{diff}(\Pi_1[x \mapsto e_x]) \cup \mathit{diff}(\Pi_2[x \mapsto e_x]).\end{gathered}$$

- (**let**). Π takes the form

$$\dfrac{\begin{array}{c}\vdots\,\Pi'\\ (\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow (\Delta, x_i \mapsto e_i')_{i=1}^n : v\end{array}}{\Gamma : \mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,e \Downarrow \Delta : v}\,(\textbf{let}).$$

By inductive hypothesis

$$(\Gamma, x_i \mapsto e_i)_{i=1}^n[x \mapsto e_x] : e \Downarrow_{\Pi'[x \mapsto e_x]} (\Delta, x_i \mapsto e_i')_{i=1}^n[x \mapsto e_x] : v$$

which, by Lemma 6.14, implies

$$(\Gamma[x \mapsto e_x], x_i \mapsto e_i)_{i=1}^n : e \Downarrow_{\Pi'[x \mapsto e_x]} (\Delta[x \mapsto e_x], x_i \mapsto e_i')_{i=1}^n : v.$$

Therefore

$$\dfrac{\begin{array}{c}\vdots\,\Pi'[x \mapsto e_x]\\ (\Gamma[x \mapsto e_x], x_i \mapsto e_i)_{i=1}^n : e \Downarrow (\Delta[x \mapsto e_x], x_i \mapsto e_i')_{i=1}^n : v\end{array}}{\Gamma[x \mapsto e_x] : \mathsf{let}\,\{x_i{=}e_i\}_{i=1}^n\,\mathsf{in}\,e \Downarrow \Delta[x \mapsto e_x] : v}\,(\textbf{let})$$

which is obviously $\Pi[x \mapsto e_x]$. Furthermore, by inductive hypothesis

$$\begin{aligned} \mathit{diff}(\Pi'[x \mapsto e_x]) &= \mathit{diff}(\Pi') \\ V(\Pi'[x \mapsto e_x]) &= V(\Pi'). \end{aligned}$$

The result follows by noting that

$$\begin{aligned} V(\Pi[x \mapsto e_x]) &= V(\Pi'[x \mapsto e_x]) \setminus \{x_i\}_{i=1}^n \\ \mathit{diff}(\Pi[x \mapsto e_x]) &= \mathit{diff}(\Pi'[x \mapsto e_x]) \setminus \{x_i\}_{i=1}^n. \end{aligned}$$

□

A.2 Lemma 6.17

Lemma 6.17. Suppose that $x \notin \mathit{dom}(\Gamma)$ and $\Gamma : e \Downarrow_\Pi \Delta : v$. Then, $(\Gamma, x \mapsto e_x) : e \Downarrow_{\Pi^+} (\Delta, x \mapsto e_x) : v$ where

$$\begin{aligned} \Pi &= \Pi^+|_{\mathit{dom}(\Gamma)} \\ V(\Pi^+) &= V(\Pi) \\ \mathit{diff}(\Pi^+) &= \mathit{diff}(\Pi). \end{aligned}$$

Proof. For this proof, without any loss of generality we rename the let-bound variables in Π to avoid accidental clash with x. We proceed by induction on Π based on its final rule:

- **(lam)**. Π takes the form

$$\frac{}{\Gamma : \lambda y.e \Downarrow \Gamma : \lambda y.e}\ \textbf{(lam)}$$

whilst Π^+ takes the form

$$\frac{}{(\Gamma, x \mapsto e_x) : \lambda y.e \Downarrow (\Gamma, x \mapsto e_x) : \lambda y.e}\ \textbf{(lam)}.$$

The result is immediate by noticing that $V(\Pi) = V(\Pi^+) = \mathit{diff}(\Pi) = \mathit{diff}(\Pi^+) = \varnothing$.

- **($\mathbf{var_y}$)** for y other than x. Π takes the form

$$\frac{\begin{array}{c}\vdots\ \Pi' \\ \Gamma' : e_y \Downarrow \Delta' : v_y\end{array}}{(\Gamma', y \mapsto e_y) : y \Downarrow (\Delta', y \mapsto v_y) : v_y}\ \textbf{(var}_\mathbf{y}\textbf{)}$$

where $(\Gamma', y \mapsto e_y) = \Gamma$ and $(\Delta', y \mapsto v_y) = \Delta$. Given that $x \notin \mathit{dom}(\Gamma')$, we can apply the inductive hypothesis to conclude $(\Gamma', x \mapsto e_x) : e_y \Downarrow_{\Pi'^+} (\Delta', x \mapsto e_x) : v_y$. Therefore, Π^+ is derivable with the form

$$\frac{\begin{array}{c}\vdots\ \Pi'^+ \\ (\Gamma', x \mapsto e_x) : e_y \Downarrow (\Delta', x \mapsto e_x) : v_y\end{array}}{(\Gamma, x \mapsto e_x) : y \Downarrow (\Delta, x \mapsto e_x) : v_y}\ \textbf{(var}_\mathbf{y}\textbf{)}.$$

$V(\Pi^+) = V(\Pi)$ follows by inductive hypothesis because $V(\Pi) = V(\Pi') \cup \{y\}$ and $V(\Pi^+) = V(\Pi'^+) \cup \{y\}$. By inductive hypothesis, $\mathit{diff}(\Pi'^+) = \mathit{diff}(\Pi')$. On the other hand,

  - if $y \in \mathit{diff}(\Pi)$, then $\mathit{diff}(\Pi) = \mathit{diff}(\Pi') \cup \{y\}$ and $\mathit{diff}(\Pi^+) = \mathit{diff}(\Pi'^+) \cup \{y\}$.
  - if $y \notin \mathit{diff}(\Pi)$, then $\mathit{diff}(\Pi) = \mathit{diff}(\Pi')$ and $\mathit{diff}(\Pi^+) = \mathit{diff}(\Pi'^+)$.

Both cases imply $\mathit{diff}(\Pi^+) = \mathit{diff}(\Pi)$, as desired.

- **(app)**. Π takes the form

$$\frac{\begin{array}{cc}\vdots\ \Pi_1 & \vdots\ \Pi_2 \\ \Gamma : e \Downarrow \Theta : \lambda z.e' & \Theta : e'[y/z] \Downarrow \Delta : v\end{array}}{\Gamma : e\ y \Downarrow \Delta : v}\ \textbf{(app)}.$$

By inductive hypothesis, $(\Gamma, x \mapsto e_x) : e \Downarrow_{\Pi_1^+} (\Theta, x \mapsto e_x) : \lambda z.e'$ and $(\Theta, x \mapsto e_x) : e'[y/z] \Downarrow_{\Pi_2^+} (\Delta, x \mapsto e_x) : v$ where

$$\begin{array}{lll} \Pi_1 & = \Pi_1^+|_{\mathit{dom}(\Gamma)} & \qquad \Pi_2 \quad = \Pi_2^+|_{\mathit{dom}(\Theta)} \\ V(\Pi_1^+) & = V(\Pi_1) & \qquad V(\Pi_2^+) = V(\Pi_2) \\ \mathit{diff}(\Pi_1^+) & = \mathit{diff}(\Pi_1) & \qquad \mathit{diff}(\Pi_2^+) = \mathit{diff}(\Pi_2). \end{array}$$

Therefore, Π^+ is derivable with the form

$$\dfrac{\begin{array}{cc}\vdots\,\Pi_1^+ & \vdots\,\Pi_2^+\\ (\Gamma, x \mapsto e_x) : e \Downarrow (\Theta, x \mapsto e_x) : \lambda z.e' & (\Theta, x \mapsto e_x) : e'[y/z] \Downarrow (\Delta, x \mapsto e_x) : v\end{array}}{(\Gamma, x \mapsto e_x) : e\ y \Downarrow (\Delta, x \mapsto e_x) : v}\ (\textbf{app}).$$

Note that $V(\Pi^+) = V(\Pi_1^+) \cup V(\Pi_2^+)$, $\mathit{diff}(\Pi^+) = \mathit{diff}(\Pi_1^+) \cup \mathit{diff}(\Pi_2^+)$, and, by Lemma 3.7, $\mathit{dom}(\Gamma) = \mathit{dom}(\Theta)$. The result is immediate.

- (**let**). Π takes the form

$$\dfrac{\begin{array}{c}\vdots\,\Pi'\\ (\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow (\Delta, x_i \mapsto e_i')_{i=1}^n : v\end{array}}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : v}\ (\textbf{let}).$$

Given that $\mathit{dom}(\Gamma) = \mathit{dom}((\Gamma, x_i \mapsto e_i)_{i=1}^n) \setminus \{x_i\}_{i=1}^n$, the assumption $x \notin \mathit{dom}(\Gamma)$ implies $x \notin \mathit{dom}((\Gamma, x_i \mapsto e_i)_{i=1}^n)$. By inductive hypothesis, $(\Gamma, x_i \mapsto e_i, x \mapsto e_x)_{i=1}^n : e \Downarrow_{\Pi'^+} (\Delta, x_i \mapsto e_i', x \mapsto e_x)_{i=1}^n : v$ where

$$\begin{array}{ll}\Pi' & = \Pi'^+|_{\mathit{dom}((\Gamma, x_i \mapsto e_i)_{i=1}^n)}\\ V(\Pi'^+) & = V(\Pi')\\ \mathit{diff}(\Pi'^+) & = \mathit{diff}(\Pi').\end{array}$$

Thus, Π^+ is derivable with the form

$$\dfrac{\begin{array}{c}\vdots\,\Pi'^+\\ (\Gamma, x_i \mapsto e_i, x \mapsto e_x)_{i=1}^n : e \Downarrow (\Delta, x_i \mapsto e_i', x \mapsto e_x)_{i=1}^n : v\end{array}}{(\Gamma, x \mapsto e_x) : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow (\Delta, x \mapsto e_x) : v}\ (\textbf{let}).$$

The result follows by noting that

$$\begin{array}{ll}\mathit{dom}(\Gamma) & = \mathit{dom}((\Gamma, x_i \mapsto e_i)_{i=1}^n) \setminus \{x_i\}_{i=1}^n\\ \mathit{dom}((\Gamma, x \mapsto e_x)) & = \mathit{dom}((\Gamma, x_i \mapsto e_i, x \mapsto e_x)_{i=1}^n) \setminus \{x_i\}_{i=1}^n\end{array}$$

$$\begin{array}{ll}V(\Pi) & = V(\Pi') \setminus \{x_i\}_{i=1}^n\\ V(\Pi^+) & = V(\Pi'^+) \setminus \{x_i\}_{i=1}^n\end{array}$$

$$\begin{array}{ll}\mathit{diff}(\Pi) & = \mathit{diff}(\Pi') \setminus \{x_i\}_{i=1}^n\\ \mathit{diff}(\Pi^+) & = \mathit{diff}(\Pi'^+) \setminus \{x_i\}_{i=1}^n.\end{array}$$

Note that by the renaming policy taken for this proof, name clash between x and x_i's is now impossible.

- (**seq**). Π takes the form

$$\dfrac{\begin{array}{cc}\vdots\,\Pi_1 & \vdots\,\Pi_2\\ \Gamma : e_1 \Downarrow \Theta : v_1 & \Theta : e_2 \Downarrow \Delta : v_2\end{array}}{\Gamma : e_1\ \mathsf{seq}\ e_2 \Downarrow \Delta : v_2}\ (\textbf{seq}).$$

By Lemma 3.7, $x \in dom(\Gamma)$ implies $x \in dom(\Theta)$. So, by inductive hypothesis, $(\Gamma, x \mapsto e_x) : e_1 \Downarrow_{\Pi_1^+} (\Theta, x \mapsto e_x) : v_1$ and $(\Theta, x \mapsto e_x) : e_2 \Downarrow_{\Pi_2^+} (\Delta, x \mapsto e_x) : v_2$ where

$$\begin{array}{lll} \Pi_1 & = \Pi_1^+|_{dom(\Gamma)} & \qquad \Pi_2 & = \Pi_2^+|_{dom(\Theta)} \\ V(\Pi_1^+) & = V(\Pi_1) & \qquad V(\Pi_2^+) & = V(\Pi_2) \\ \mathit{diff}(\Pi_1^+) & = \mathit{diff}(\Pi_1) & \qquad \mathit{diff}(\Pi_2^+) & = \mathit{diff}(\Pi_2). \end{array}$$

Therefore, Π^+ is derivable with the form

$$\frac{\begin{array}{cc} \vdots\, \Pi_1^+ & \vdots\, \Pi_2^+ \\ (\Gamma, x \mapsto e_x) : e_1 \Downarrow (\Theta, x \mapsto e_x) : v_1 & (\Theta, x \mapsto e_x) : e_2 \Downarrow (\Delta, x \mapsto e_x) : v_2 \end{array}}{(\Gamma, x \mapsto e_x) : e_1 \;\mathsf{seq}\; e_2 \Downarrow (\Delta, x \mapsto e_x) : v_2}\ \textbf{(seq)}.$$

Note that $V(\Pi^+) = V(\Pi_1^+) \cup V(\Pi_2^+)$, $\mathit{diff}(\Pi^+) = \mathit{diff}(\Pi_1^+) \cup \mathit{diff}(\Pi_2^+)$, and, by Lemma 3.7, $dom(\Gamma) = dom(\Theta)$. The result is immediate.

□

Appendix B

Additional Heap Proofs

This chapter contains the proofs we postponed in Chapter 7. This includes Theorem 7.10, and Theorem 7.11, which are presented in Sections B.1 and B.2, respectivey.

B.1 Theorem 7.10

Theorem 7.10. Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$ and $x \in V(\Pi)$.

Suppose $x \in \mathit{atomic}(\Gamma)$; so in particular $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ for some v_x. Then $\Delta(x) = v_x$ and

$$\Gamma[x \mapsto v_x] : e \Downarrow_{\Pi[x \mapsto v_x]} \Delta : v.$$

Furthermore,

$$\begin{aligned} \mathit{diff}(\Pi[x \mapsto v_x]) &= \mathit{diff}(\Pi) \setminus \{x\} \\ V(\Pi[x \mapsto v_x]) \cup V(\Pi_x) &= V(\Pi) \\ x &\in V(\Pi[x \mapsto v_x]). \end{aligned}$$

Proof. Induction on Π based on its final rule:

- (**lam**). Π takes the form

$$\frac{}{\Gamma : \lambda y.e \Downarrow \Gamma : \lambda y.e}\ (\mathbf{lam}).$$

  There is nothing to consider for this case because $x \notin V(\Pi)$. (In fact, $V(\Pi) = \varnothing$.)

- (**var$_\mathbf{x}$**). Π takes the form

$$\frac{\begin{array}{c}\vdots \\ \Gamma' : e_x \Downarrow \Delta' : v_x\end{array}}{(\Gamma', x \mapsto e_x) : x \Downarrow (\Delta', x \mapsto v_x) : v_x}\ (\mathbf{var_x})$$

where $(\Gamma', x \mapsto e_x) = \Gamma$ and $(\Delta', x \mapsto v_x) = \Delta$. (Note that, as also notified in Remark 7.6, $\Delta' = \Gamma'$ here.) By construction (Definition 7.5), in this case, $\Pi[x \mapsto v_x]$ will take the form

$$\dfrac{\dfrac{}{\Gamma' : v_x \Downarrow \Gamma' : v_x}\,(\textbf{lam})}{(\Gamma', x \mapsto v_x) : x \Downarrow (\Gamma', x \mapsto v_x) : v_x}\,(\textbf{var}_\textbf{x}).$$

Observe that, by determinism of our operational semantics (Theorem 3.9), $\Pi =_{\mathsf{let}\alpha} \Pi_x$. (See Notation 3.8 for $=_{\mathsf{let}\alpha}$.) Thus, $x \in \mathit{atomic}(\Gamma)$ implies $\mathit{diff}(\Pi) = \mathit{diff}(\Pi_x) = \{x\}$ by Lemma 7.3, and

$$\begin{gathered}\varnothing = \mathit{diff}(\Pi[x \mapsto v_x]) = \mathit{diff}(\Pi) \setminus \{x\} \\ V(\Pi[x \mapsto v_x]) \cup V(\Pi_x) = \{x\} \cup V(\Pi) = V(\Pi)\end{gathered}$$

because by assumption $x \in V(\Pi)$. Finally, note that $x \in \mathit{dom}((\Gamma', x \mapsto v_x))$ and $\Pi[x \mapsto v_x]$ contains an instance of $(\textbf{var}_\textbf{x})$. By definition (Definition 6.1), these mean that $x \in V(\Pi[x \mapsto v_x])$.

- $(\textbf{var}_\textbf{y})$ for y other than x. Π takes the form

$$\dfrac{\begin{array}{c}\vdots\,\Pi' \\ \Gamma' : e_y \Downarrow \Delta' : v_y\end{array}}{(\Gamma', y \mapsto e_y) : y \Downarrow (\Delta', y \mapsto v_y) : v_y}\,(\textbf{var}_\textbf{y})$$

where $\Gamma = (\Gamma', y \mapsto e_y)$ and $\Delta = (\Delta', y \mapsto v_y)$. By assumption $x \in V(\Pi)$ which implies $x \in V(\Pi')$. It follows by Lemma 7.7 that $y \notin V(\Pi_x)$. Thus, by Lemma 7.9, $x \in \mathit{atomic}(\Gamma')$ and $\Gamma' : x \Downarrow \Gamma'[x \mapsto v_x] : v_x$. Namely, $\Pi'[x \mapsto v_x]$ satisfies all the properties given in the lemma. By inductive hypothesis, then $\Delta'(x) = v_x$ and $\Gamma'[x \mapsto v_x] : e_y \Downarrow_{\Pi'[x \mapsto v_x]} \Delta' : v_y$. Hence, $\Delta(x) = v_x$ and

$$(\Gamma'[x \mapsto v_x], y \mapsto e_y) : y \Downarrow_{\Pi[x \mapsto v_x]} \Delta : v_y. \tag{B.1}$$

By Lemma 6.14,

$$(\Gamma'[x \mapsto v_x], y \mapsto e_y) = (\Gamma', y \mapsto e_y)[x \mapsto v_x]. \tag{B.2}$$

We use (B.1) and (B.2) to observe that $\Gamma[x \mapsto v_x] : y \Downarrow_{\Pi[x \mapsto v_x]} \Delta : v_y$.

$$\begin{gathered}\mathit{diff}(\Pi[x \mapsto v_x]) = \mathit{diff}(\Pi) \setminus \{x\} \\ V(\Pi[x \mapsto v_x]) \cup V(\Pi_x) = V(\Pi)\end{gathered}$$

follow by inductive hypothesis and

$$\begin{aligned}V(\Pi) &= V(\Pi') \cup \{y\} \\ \mathit{diff}(\Pi) &= \mathit{diff}(\Pi') \cup \{y\}.\end{aligned}$$

- **(app)**. Π takes the form:

$$\frac{\begin{array}{cc}\vdots\,\Pi_1 & \vdots\,\Pi_2\\ \Gamma : e' \Downarrow \Theta : \lambda z.e'' & \Theta : e''[y/z] \Downarrow \Delta : v\end{array}}{\Gamma : e'\ y \Downarrow \Delta : v}\ \textbf{(app)}.$$

  There are now two cases:

  - The case that $x \in V(\Pi_1)$. By inductive hypothesis $\Theta(x) = v_x$ and $\Gamma[x \mapsto v_x] : e' \Downarrow_{\Pi_1[x \mapsto v_x]} \Theta : \lambda z.e''$. By Lemma 3.10, $\Delta(x) = \Theta(x) = v_x$. Given that $\Theta(x) = v_x$, by Lemma 6.13, $\Pi_2[x \mapsto v_x] = \Pi_2$. Thus, by the **(app)** case of Definition 7.5, we can combine $\Pi_1[x \mapsto v_x]$ with Π_2 to get $\Pi[x \mapsto v_x]$ and observe that $\Gamma[x \mapsto v_x] : e'y \Downarrow_{\Pi[x \mapsto v_x]} \Delta : v$. Finally, by definition (Definition 6.1), $x \in V(\Pi_1[x \mapsto v_x])$, implies $x \in V(\Pi[x \mapsto v_x])$. The result follows.
  - The case that $x \notin V(\Pi_1)$. It follows that $x \in V(\Pi_2)$. By Lemma 6.15, $\Gamma[x \mapsto v_x] : e' \Downarrow_{\Pi_1[x \mapsto v_x]} \Theta[x \mapsto v_x] : \lambda z.e''$. By Lemma 7.8, $x \in \mathit{atomic}(\Theta)$, and $\Theta : x \Downarrow \Theta_x : v_x$ for some Θ_x. Therefore, by inductive hypothesis $\Delta(x) = v_x$ and $\Theta[x \mapsto v_x] : e''[y/z] \Downarrow_{\Pi_2[x \mapsto v_x]} \Delta : v$. We combine $\Pi_1[x \mapsto v_x]$ with $\Pi_2[x \mapsto v_x]$ to get $\Pi[x \mapsto v_x]$ and observe that $\Gamma[x \mapsto v_x] : e'y \Downarrow_{\Pi[x \mapsto v_x]} \Delta : v$. (Note that, by the **(app)** case of Definition 7.5, this combination is indeed $\Pi[x \mapsto v_x]$.) Finally, by definition (Definition 6.1), $x \in V(\Pi_2[x \mapsto v_x])$, implies $x \in V(\Pi[x \mapsto v_x])$. The result follows.

  In both cases, by inductive hypothesis,

$$V(\Pi_1[x \mapsto v_x]) \cup V(\Pi_x) = V(\Pi_1) \tag{B.3}$$
$$V(\Pi_2[x \mapsto v_x]) \cup V(\Pi_x) = V(\Pi_2) \tag{B.4}$$
$$\mathit{diff}(\Pi_1[x \mapsto v_x]) = \mathit{diff}(\Pi_1) \setminus \{x\} \tag{B.5}$$
$$\mathit{diff}(\Pi_2[x \mapsto v_x]) = \mathit{diff}(\Pi_2) \setminus \{x\}. \tag{B.6}$$

  On the other hand, by Definitions 6.1 and 6.4, respectively,

$$V(\Pi) = V(\Pi_1) \cup V(\Pi_2) \tag{B.7}$$
$$\mathit{diff}(\Pi) = \mathit{diff}(\Pi_1) \cup \mathit{diff}(\Pi_2). \tag{B.8}$$

  By substituting (B.3) and (B.4) into (B.7), we get

$$V(\Pi[x \mapsto v_x]) \cup V(\Pi_x) = V(\Pi).$$

  Similarly, by substituting (B.5) and (B.6) into (B.8), we get

$$\mathit{diff}(\Pi[x \mapsto v_x]) = \mathit{diff}(\Pi) \setminus \{x\},$$

  as desired.

- **(let)**. Π takes the form

$$\frac{\begin{array}{c}\vdots\,\Pi'\\ (\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow (\Delta, x_i \mapsto e_i')_{i=1}^n : v\end{array}}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : v}\ \textbf{(let)}.$$

  Observe first that $x \in \mathit{atomic}(\Gamma)$ implies $x \in \mathit{atomic}((\Gamma, x_i \mapsto e_i)_{i=1}^n)$ by Lemma 7.9. Secondly, $x \in V(\Pi)$ implies $x \in V(\Pi')$. Therefore, by inductive hypothesis:

  - $(\Gamma, x_i \mapsto e_i)_{i=1}^n[x \mapsto v_x] : e \Downarrow_{\Pi'[x \mapsto v_x]} (\Delta, x_i \mapsto e_i')_{i=1}^n : v$ which, by Lemma 6.14, can be rewritten as $(\Gamma[x \mapsto v_x], x_i \mapsto e_i)_{i=1}^n : e \Downarrow_{\Pi'[x \mapsto v_x]} (\Delta, x_i \mapsto e_i')_{i=1}^n : v$. Extending with **(let)**, we deduce $\Gamma[x \mapsto v_x] : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow_{\Pi[x \mapsto v_x]} \Delta : v$. By the **(let)** case of Definition 7.5, we know that this combination is indeed $\Pi[x \mapsto v_x]$, as desired.
  - $(\Delta, x_i \mapsto e_i')_{i=1}^n(x) = v_x$ which, given that x_i's are fresh, implies $\Delta(x) = v_x$.

  Given that $\{x_i\}_{i=1}^n \cap \mathit{dom}(\Gamma) = \varnothing$, by Definitions 6.1 and 6.4, respectively:

$$V(\Pi) = V(\Pi') \setminus \{x_i\}_{i=1}^n \tag{B.9}$$
$$V(\Pi[x \mapsto v_x]) = V(\Pi'[x \mapsto v_x]) \setminus \{x_i\}_{i=1}^n \tag{B.10}$$
$$\mathit{diff}(\Pi) = \mathit{diff}(\Pi') \setminus \{x_i\}_{i=1}^n \tag{B.11}$$
$$\mathit{diff}(\Pi[x \mapsto v_x]) = \mathit{diff}(\Pi'[x \mapsto v_x]) \setminus \{x_i\}_{i=1}^n. \tag{B.12}$$

  On the other hand, by inductive hypothesis,

$$V(\Pi'[x \mapsto v_x]) \cup V(\Pi_x) = V(\Pi') \tag{B.13}$$
$$\mathit{diff}(\Pi'[x \mapsto v_x]) = \mathit{diff}(\Pi') \setminus \{x\}. \tag{B.14}$$

  By substituting (B.9) and (B.10) into (B.13), we get

$$V(\Pi[x \mapsto v_x]) \cup V(\Pi_x) = V(\Pi).$$

  Similarly, by substituting (B.11) and (B.12) into (B.14), we get

$$\mathit{diff}(\Pi'[x \mapsto v_x]) = \mathit{diff}(\Pi') \setminus \{x\},$$

  as desired.

- **(seq)**. Π takes the form

$$\frac{\begin{array}{cc}\vdots\,\Pi_1 & \vdots\,\Pi_2\\ \Gamma : e_1 \Downarrow \Theta : v_1 & \Theta : e_2 \Downarrow \Delta : v_2\end{array}}{\Gamma : e_1\ \mathsf{seq}\ e_2 \Downarrow \Delta : v_2}\ \textbf{(seq)}.$$

  There are now two cases:

– The case that $x \in V(\Pi_1)$. By inductive hypothesis $\Theta(x) = v_x$ and $\Gamma[x \mapsto v_x] : e_1 \Downarrow_{\Pi_1[x \mapsto v_x]} \Theta : v_1$. By Lemma 3.10, $\Delta(x) = \Theta(x) = v_x$. Given that $\Theta(x) = v_x$, by Lemma 6.13, $\Pi_2[x \mapsto v_x] = \Pi_2$. Thus, by the **(app)** case of Definition 7.5, we can combine $\Pi_1[x \mapsto v_x]$ with Π_2 to get $\Pi[x \mapsto v_x]$ and observe that $\Gamma[x \mapsto v_x] : e_1 \ \mathsf{seq}\ e_2 \Downarrow_{\Pi[x \mapsto v_x]} \Delta : v_2$. Finally, by definition (Definition 6.1), $x \in V(\Pi_1[x \mapsto v_x])$, implies $x \in V(\Pi[x \mapsto v_x])$. The result follows.

– The case that $x \notin V(\Pi_1)$. It follows that $x \in V(\Pi_2)$. By Lemma 6.15, $\Gamma[x \mapsto v_x] : e_1 \Downarrow_{\Pi_1[x \mapsto v_x]} \Theta[x \mapsto v_x] : v_1$. By Lemma 7.8, $x \in \mathit{atomic}(\Theta)$, and $\Theta : x \Downarrow \Theta_x : v_x$ for some Θ_x. Therefore, by inductive hypothesis $\Delta(x) = v_x$ and $\Theta[x \mapsto v_x] : e_2 \Downarrow_{\Pi_2[x \mapsto v_x]} \Delta : v_2$. We combine $\Pi_1[x \mapsto v_x]$ with $\Pi_2[x \mapsto v_x]$ to get $\Pi[x \mapsto v_x]$ and observe that $\Gamma[x \mapsto v_x] : e_1 \ \mathsf{seq}\ e_2 \Downarrow_{\Pi[x \mapsto v_x]} \Delta : v_2$. (Note that, by the **(app)** case of Definition 7.5, this combination is indeed $\Pi[x \mapsto v_x]$.) Finally, by definition (Definition 6.1), $x \in V(\Pi_2[x \mapsto v_x])$, implies $x \in V(\Pi[x \mapsto v_x])$. The result follows.

In both cases, by inductive hypothesis,

$$V(\Pi_1[x \mapsto v_x]) \cup V(\Pi_x) = V(\Pi_1) \tag{B.15}$$
$$V(\Pi_2[x \mapsto v_x]) \cup V(\Pi_x) = V(\Pi_2) \tag{B.16}$$
$$\mathit{diff}(\Pi_1[x \mapsto v_x]) = \mathit{diff}(\Pi_1) \setminus \{x\} \tag{B.17}$$
$$\mathit{diff}(\Pi_2[x \mapsto v_x]) = \mathit{diff}(\Pi_2) \setminus \{x\}. \tag{B.18}$$

On the other hand, by Definitions 6.1 and 6.4, respectively,

$$V(\Pi) = V(\Pi_1) \cup V(\Pi_2) \tag{B.19}$$
$$\mathit{diff}(\Pi) = \mathit{diff}(\Pi_1) \cup \mathit{diff}(\Pi_2). \tag{B.20}$$

By substituting (B.15) and (B.16) into (B.19), we get

$$V(\Pi[x \mapsto v_x]) \cup V(\Pi_x) = V(\Pi).$$

Similarly, by substituting (B.17) and (B.18) into (B.20), we get

$$\mathit{diff}(\Pi[x \mapsto v_x]) = \mathit{diff}(\Pi) \setminus \{x\},$$

as desired.

□

B.2 Theorem 7.11

Theorem 7.11. Suppose $x \in \mathit{atomic}(\Gamma)$; so in particular $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ for some v_x, and Π_x such that $\mathit{diff}(\Pi_x) = \{x\}$. Suppose $\Gamma[x \mapsto v_x] : e \Downarrow_{\Pi'} \Delta : v$ and $x \in V(\Pi')$. Then $\Gamma : e \Downarrow_{\Pi} \Delta : v$ for a Π such that $\Pi[x \mapsto v_x] = \Pi'$ and $x \in V(\Pi)$.

Proof. Write $e_x = \Gamma(x)$. We work by induction on Π' based on its final rule:

- **(lam)**. Π' takes the form

$$\frac{}{\Gamma : \lambda y.e \Downarrow \Gamma : \lambda y.e}\ \textbf{(lam)}.$$

  There is nothing to consider for this case because $x \notin V(\Pi')$. (In fact, $V(\Pi') = \varnothing$.)

- **($\mathbf{var_x}$)**. Π' takes the form

$$\frac{\dfrac{}{\Gamma' : v_x \Downarrow \Gamma' : v_x}\ \textbf{(lam)}}{(\Gamma', x \mapsto v_x) : x \Downarrow (\Gamma', x \mapsto v_x) : v_x}\ \textbf{($\mathbf{var_x}$)}$$

  where $\Gamma = (\Gamma', x \mapsto v_x) = \Delta$. We take $\Pi = \Pi_x$. By Definition 7.5, as $x \in \mathit{atomic}(\Gamma)$, we have $\Pi_x[x \mapsto v_x] = \Pi'$.

- **($\mathbf{var_y}$)** for y other than x. Π' takes the form

$$\frac{\begin{array}{c}\vdots\ \Pi'_y \\ \Gamma'[x \mapsto v_x] : e_y \Downarrow \Delta' : v_y\end{array}}{(\Gamma', y \mapsto e_y)[x \mapsto v_x] : y \Downarrow (\Delta', y \mapsto v_y) : v_y}\ \textbf{($\mathbf{var_y}$)}$$

  where $\Gamma = (\Gamma', y \mapsto e_y)$ and $\Delta = (\Delta', y \mapsto v_y)$. By assumption $x \in V(\Pi')$. It follows by Lemma 7.7 that $y \notin V(\Pi_x)$. Therefore, by Lemma 7.9, $x \in \mathit{atomic}(\Gamma')$, and $\Gamma' : x \Downarrow \Xi : v_x$ for some heap Ξ. By inductive hypothesis,

$$\Gamma' : e_y \Downarrow_{\Pi_y} \Delta' : v_y \text{ for some } \Pi_y \text{ such that } \Pi_y[x \mapsto v_x] = \Pi'_y, \text{ and} \quad \text{(B.21)}$$
$$x \in V(\Pi_y). \quad \text{(B.22)}$$

  We extend Π_y with a **($\mathbf{var_y}$)** to get Π. By (B.21), we observe that $\Gamma : y \Downarrow_\Pi \Delta : v_y$ such that $\Pi[x \mapsto v_x] = \Pi'$. By definition (Definition 6.1), given that $x \in \mathit{dom}(\Gamma) \cap \mathit{dom}(\Gamma')$, (B.22) implies $x \in V(\Pi)$, as desired.

- **(app)**. Π' takes the form

$$\frac{\begin{array}{cc}\vdots\ \Pi'_1 & \vdots\ \Pi'_2 \\ \Gamma[x \mapsto v_x] : e' \Downarrow \Theta : \lambda z.e'' & \Theta : e''[y/z] \Downarrow \Delta : v\end{array}}{\Gamma[x \mapsto v_x] : e'y \Downarrow \Delta : v}\ \textbf{(app)}.$$

  There are now two cases:

  - The case that $x \in V(\Pi'_1)$. By inductive hypothesis $\Gamma : e' \Downarrow_{\Pi_1} \Theta : \lambda z.e''$ and $\Pi_1[x \mapsto v_x] = \Pi'_1$. Furthermore, by inductive hypothesis, $x \in V(\Pi_1)$, which by Corollary 6.8, implies $\Theta(x)$ is a value. Therefore, by Lemma 6.13, $\Pi'_2[x \mapsto v_x] = \Pi'_2$. Hence, by the **(app)** case of Definition 7.5, we can combine Π_1 with Π'_2 to construct a Π such that $\Gamma : e'y \Downarrow_\Pi \Delta : v$ and $\Pi[x \mapsto v_x] = \Pi'$. The result follows because, by definition (Definition 6.1), $x \in V(\Pi_1)$ implies $x \in V(\Pi)$, as desired.

– The case that $x \notin V(\Pi_1')$. It follows that $x \in V(\Pi_2')$. By Lemma 6.15, $\Gamma : e' \Downarrow_{\Pi_1'[x \mapsto e_x]} \Theta[x \mapsto e_x] : \lambda z.e''$. By Lemma 7.8, $x \in \mathit{atomic}(\Theta[x \mapsto e_x])$. By inductive hypothesis, $\Theta[x \mapsto e_x] : e''[y/z] \Downarrow_{\Pi_2} \Delta : v$ for a Π_2 such that $\Pi_2[x \mapsto v_x] = \Pi_2'$ and $x \in V(\Pi_2)$. By construction (the **(app)** case of Definition 7.5), we can combine $\Pi_1'[x \mapsto e_x]$ and Π_2 to get a derivation Π such that $\Pi[x \mapsto v_x] = \Pi'$. The result follows because, by definition (Definition 6.1), $x \in V(\Pi_2)$ implies $x \in V(\Pi)$, as desired.

- **(let)**. Π' takes the form

$$\dfrac{\begin{array}{c}\vdots\,\Pi_l' \\ (\Gamma[x \mapsto v_x], x_i \mapsto e_i)_{i=1}^n : e \Downarrow (\Delta, x_i \mapsto e_i')_{i=1}^n : v\end{array}}{\Gamma[x \mapsto v_x] : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : v}\ \textbf{(let)}.$$

Observe that, by Lemma 6.14, $(\Gamma[x \mapsto v_x], x_i \mapsto e_i)_{i=1}^n = (\Gamma, x_i \mapsto e_i)_{i=1}^n[x \mapsto v_x]$. Thus, $(\Gamma, x_i \mapsto e_i)_{i=1}^n[x \mapsto v_x] : e \Downarrow_{\Pi_l'} (\Delta, x_i \mapsto e_i')_{i=1}^n : v$. On the other hand, given that x_i's are fresh, by Lemma 7.9, $x \in \mathit{atomic}(\Gamma)$ implies $x \in \mathit{atomic}((\Gamma, x_i \mapsto e_i)_{i=1}^n)$. Therefore, by inductive hypothesis, $(\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow_{\Pi_l} (\Delta, x_i \mapsto e_i')_{i=1}^n : v$ for some Π_l such that $\Pi_l[x \mapsto v_x] = \Pi_l'$. We extend Π_l with **(let)** to get Π such that $\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow_{\Pi} \Delta : v$ and $\Pi[x \mapsto v_x] = \Pi'$.

- **(seq)**. Π' takes the form

$$\dfrac{\begin{array}{cc}\vdots\,\Pi_1' & \vdots\,\Pi_2' \\ \Gamma[x \mapsto v_x] : e_1 \Downarrow \Theta : v_1 & \Theta : e_2 \Downarrow \Delta : v_2\end{array}}{\Gamma[x \mapsto v_x] : e_1\ \mathsf{seq}\ e_2 \Downarrow \Delta : v_2}\ \textbf{(seq)}.$$

There are now two cases:

– The case that $x \in V(\Pi_1')$. By inductive hypothesis $\Gamma : e_1 \Downarrow_{\Pi_1} \Theta : v_1$ and $\Pi_1[x \mapsto v_x] = \Pi_1'$. Furthermore, by inductive hypothesis, $x \in V(\Pi_1)$, which by Corollary 6.8, implies $\Theta(x)$ is a value. Therefore, by Lemma 6.13, $\Pi_2'[x \mapsto v_x] = \Pi_2'$. Hence, by the **(seq)** case of Definition 7.5, we can combine Π_1 with Π_2' to construct a Π such that $\Gamma : e_1\ \mathsf{seq}\ e_2 \Downarrow_{\Pi} \Delta : v_2$ and $\Pi[x \mapsto v_x] = \Pi'$. The result follows because, by definition (Definition 6.1), $x \in V(\Pi_1)$ implies $x \in V(\Pi)$, as desired.

– The case that $x \notin V(\Pi_1')$. It follows that $x \in V(\Pi_2')$. By Lemma 6.15, $\Gamma : e_1 \Downarrow_{\Pi_1'[x \mapsto e_x]} \Theta[x \mapsto e_x] : v_1$. By Lemma 7.8, $x \in \mathit{atomic}(\Theta[x \mapsto e_x])$. By inductive hypothesis, $\Theta[x \mapsto e_x] : e_2 \Downarrow_{\Pi_2} \Delta : v_2$ for a Π_2 such that $\Pi_2[x \mapsto v_x] = \Pi_2'$ and $x \in V(\Pi_2)$. By construction (the **(seq)** case of Definition 7.5), we can combine $\Pi_1'[x \mapsto e_x]$ and Π_2 to get a derivation Π such that $\Pi[x \mapsto v_x] = \Pi'$. The result follows because, by definition (Definition 6.1), $x \in V(\Pi_2)$ implies $x \in V(\Pi)$, as desired.

□

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] S. Abramsky. The Lazy Lambda Calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison Wesley, 1990.

[3] S. Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105(2):159–267, August 1993.

[4] P. Achten, M. van Eekelen, M. de Mol, and R. Plasmeijer. A Common Arrow Based Semantics for GEC and iData Applications. Technical Report ICIS–R08023, Radboud University Nijmegen, December 2008.

[5] P. Achten, M. van Eekelen, and M. Plasmeijer. Compositional model-views with generic graphical user interfaces. In B. Jayaraman, editor, *PADL*, volume 3057 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2004.

[6] P. Achten, M. van Eekelen, R. Plasmeijer, and A. van Weelden. Automatic generation of editors for higher-order data structures. In W.-N. Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 262–279. Springer, 2004.

[7] Z. Ariola and M. Felleisen. The call-by-need λ-calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.

[8] Z. Ariola and J. Klop. Cyclic lambda graph rewriting. In *Proceedings of the 8th IEEE Symposium on Logic in Computer Science*, Paris, 1994.

[9] Z. Ariola and J. Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3–4):207–240, 1996.

[10] Z. Ariola, J. Maraist, M. Odersky, M. Felleisen, and P. Wadler. A call-by-need lambda calculus. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246, New York, NY, USA, 1995. ACM.

[11] C. Baker-Finch, D. King, and P. Trinder. An operational semantics for parallel lazy evaluation. In *International Conference on Functional Programming, Montreal (ICFP)*, pages 162–173, sep 2000.

[12] H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics (revised ed.)*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. http://www.andrew.cmu.edu/user/cebrown/notes/barendregt.html.

[13] O. Bastonero. *Modèles fortement stables du Lambda-calcul et résultats d'incomplétude*. Ph.D. thesis, Université Paris VII, 1996.

[14] O. Bastonero and X. Gouy. Strong stability and the incompleteness of stable models for λ-calculus. *Annals of Pure and Applied Logic*, 100:247–277, 1999.

[15] O. Bastonero, A. Pravato, and S. Ronchi Della Rocca. Structures for lazy semantics. In Gries and de Roever, editors, *Programming Concepts and Methods (PROCOMET'98)*, pages 30 – 48. Chaptman & Hall, 1998.

[16] R. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 2nd edition, 1998.

[17] P. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82(2):389–402, 1991.

[18] N. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 206–217, New York, NY, USA, 2006. ACM Press.

[19] M. de Mol. Examples of theorems from 'introduction to functional programming' proved in the prototype CleanProverSystem. http://www.cs.ru.nl/~maartenm/CleanProverSystem/IFP2_Example.ps, 1998.

[20] M. de Mol. *Reasoning About Functional Programs:* Sparkle*, a proof assistant for* Clean. Ph.D. thesis, Department of Computing Science, University of Nijmegen, 2009.

[21] M. de Mol and M. van Eekelen. A proof tool dedicated to Clean. In *Proceedings of Applications of Graph Transformations With Industrial Relevance, AGTIVE'99*, 1999.

[22] M. de Mol, M. van Eekelen, and M. Plasmeijer. Theorem proving for functional programmers. In *IFL*, pages 55–71, 2001. http://link.springer.de/link/service/series/0558/bibs/2312/23120055.htm.

[23] M. de Mol, M. van Eekelen, and R. Plasmeijer. The Mathematical Foundation of the Proof Assistant Sparkle. Technical Report ICIS–R07025, Radboud University Nijmegen, November 2007.

[24] M. de Mol, M. van Eekelen, and R. Plasmeijer. Proving properties of lazy functional programs with Sparkle. In Zoltán Horváth, editor, *2nd Central-European Functional Programming School, CEFP 2007*, LNCS Tutorial Series, Cluj-Napoca, Romania, 2008. Springer. To appear.

[25] M. de Mol, M. van Eekelen, and R. Plasmeijer. A single-step term-graph reduction system for proof assistants. In Albert Zündorf Andy Schürr, Manfred Nagl, editor, *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007. Proceedings of Selected and Invited Papers*, LNCS, Universität Kassel, Germany, 2008. Springer Verlag. To appear.

[26] M. Draghicescu and S. Purushothaman. A uniform treatment of order of evaluation and aggregate update. 118(2):231–262, September 1993.

[27] J. Fairbairn and S. Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In *Proceedings of the 1987 Functional Programming and Computer Architecture Conference, Portland, Oregon*, pages 34–45, September 1987.

[28] M. A. A. Ghaly, S. S. Daoud, A. A. Taha, and S. M. Aly. Operational semantics for lazy evaluation. *Journal of Computer Science*, 3(8):41–77, 2007.

[29] The Glasgow Haskell Compiler. WWW page. Available at http://www.dcs.gla.ac.uk/fp/software/ghc/.

[30] A. Gill, J. Launchbury, and S. Peyton-Jones. A short cut to deforestation. In *6th ACM SIGPLAN-SIGARCH International Conference on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, pages 223–232. ACM Press, 1993.

[31] J. Y. Girard. *Interprétation functionelle et élimination des coupures dans l'arithm étique d'ordre supérieure*. PhD thesis, Université Paris VII, 1972.

[32] A. Gordon. Bisimilarity as a theory of functional programming. In *Proceedings of 11th Conference on Mathematical Foundations of Programming Semantics*, 1995.

[33] J. Greiner and G. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Transactions on Programming Languages and Systems*, 21(2):240–285, 1999.

[34] C. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, Cambridge, MA, 1992.

[35] J. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 333–343, Philadelphia, Pennsylvania, 4–7 June 1990. IEEE Computer Society Press.

[36] J. Hall, C. Baker-Finch, P. Trinder, and D. King. Towards an operational semantics for a parallel non-strict functional language. In *Proceedings of the 10th. Int. Workshop on Implementation of Functional Languages*, volume 1595 of *Lecture Notes in Computer Science*, pages 55–67, sep 1998.

[37] W. Harrison, T. Sheard, and J. Hook. Fine control of demand in Haskell. In *Sixth International Conference on the Mathematics of Program Construction, Dagstuhl, Germany*, volume 2386 of *Lecture Notes in Computer Science*, pages 68–93. Springer Verlag, July 2002.

[38] W. L. Harrison and R. B. Kieburtz. The logic of demand in Haskell. *Journal of Functional Programming*, 15(6):837–891, 2005.

[39] P. Henderson and J. Morris. A lazy evaluator. In *3rd Annual ACM Symposium on Principles of Programming Languages*, pages 95–103. SIGPLAN and SIGACT, 1976.

[40] M. Hidalgo-Herrero. *Semánticas formales para un lenguaje funcional paralelo*. Ph.D. thesis, Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2004.

[41] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters (World Scientific Publishing Company)*, 12(2):211–228, 2002.

[42] F. Honsell and M. Lenisa. Some results on the full abstraction problem for restricted lambda calculi. In *Mathematical foundations of computer science 1993 (Gdańsk, 1993)*, pages 84–104. Springer, Berlin, 1993.

[43] J. Hook. Programatica home page, 2001. http://www.cse.ogi.edu/PacSoft/projects/programatica/default.htm.

[44] G. Huet and J. Lévy. Computations in orthogonal rewriting systems. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic—Essays in Honor of Alan Robinson*, pages 395–443. MIT Press, Cambridge, MA, 1991.

[45] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000. http://www.cs.chalmers.se/~rjmh/Papers/arrows.ps.

[46] R. Hughes. Super-combinators: A new implementation method for applicative languages. In *ACM Symposium on Lisp and Functional Programming*, August 1982.

[47] P. Johann and J. Voigtländer. Free theorems in the presence of *seq*. In N. D. Jones and X. Leroy, editors, *POPL'04: Venice, Italy, Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 39 of *SIGPLAN Notices*, pages 99–110. ACM Press, January 2004. Extended version accepted to *Fundamenta Informaticae*.

[48] P. Johann and J. Voigtländer. The impact of *seq* on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1–2):63–102, 2006.

[49] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of ACM Conference on Compiler Construction*, pages 58–69, Montreal, 1984.

[50] M. Jones and J. Peterson. Hugs 98 User Manual, 1999. Available at http://www.cse.ogi.edu/PacSoft/projects/Hugs/pages/downloading.htm.

[51] G. Kahn. Natural Semantics. In F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings of STACS*, 1987. also INRIA, RR 416, Natural Semantics on the Computer.

[52] R. Kieburtz. P-logic: property verification for Haskell programs. Technical report, OGI, 2002.

[53] P. J. Koopman, Jr. and P. Lee. A fresh look at combinator graph reduction (or, having a tigre by the tail). *SIGPLAN Notices*, 24(7):110–119, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[54] J. Kotowicz and K. Raczkowski. Coherent space. *Journal of Formalized Mathematics*, 4, 1992. http://mizar.org/JFM/Vol4/coh_sp.html.

[55] J. Launchbury. A natural semantics for lazy evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154. ACM, 1993.

[56] J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. In *European Symposium on Programming*, pages 204–218, 1996.

[57] D. Leivant. Polymorphic type inference. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 88–98, Austin, Texas, January 1983.

[58] H-W. Loidl. Glasgow Parallel Haskell. WWW page, January 2001. http://www.macs.hw.ac.uk/~dsg/gph/.

[59] G. Longo. Set theoretical models of lambda calculus: Theory, expansions and isomorphisms. *Annales of Pure and Applied Logic*, 24:153–188, 1983.

[60] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[61] J. Maraist. *Comparing Reduction Strategies in Resource-Conscious Lambda Calculi.* PhD thesis, University of Karlsruhe, Postfach 6980, D-76128 Karlsruhe, Germany, October 1996.

[62] J. Maraist, M. Odersky, and P. Wadler. The call-by-need λ-calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.

[63] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *ICFP '09: Proceeding of the 14th ACM SIGPLAN international conference on Functional programming*, Aug 2009.

[64] S. Marlow, M. Wallace, R. Paterson, and S. Kurtzberg. `seq` vs. `pseq`, 2006. http://www.mail-archive.com/glasgow-haskell-users@haskell.org/msg11022.html.

[65] J. McCarthy. A Basis for a Mathematical Theory of Computations. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[66] R. Milner. Fully abstract models of typed Lambda-Calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[67] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, MA, 1996.

[68] A. Moran. *Call-by-name, Call-by-need, and McCarthy's Amb*. PhD thesis, Department of Computing Science, Chalmers University, 1998.

[69] A. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *The 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 45–56, San Antonio, Texas, 1999.

[70] J. H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, Massachusets Institute of Technology, 1968.

[71] Murdoch J. Gabbay, Seyed H. HAERI (Hossein), Yolanda Ortega-Mallén, and Phil W. Trinder. Reasoning about selective strictness: operational equivalence, heaps and call-by-need evaluation, new inductive principles. 2009. Submitted to ICFP'09: Fouteenth ACM SIGPLAN International Conference on Functional Programming.

[72] C-H Ong. *The Lazy Lambda Calculus*. PhD thesis, Imperial College, London, 1988.

[73] C.-H. L. Ong. *The Lazy Lambda Calculus: An Investigation in the Foundations of Functional Programming*. PhD thesis, Imperial College of Science and Technology, University of London, May 1988.

[74] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS Language Reference, Version 2.4*. SRI International, 2001. http://pvs.csl.sri.com/doc/pvs-language-reference.pdf.

[75] L. Paulson. *The* Isabelle *reference manual*, April 2009. http://isabelle.in.tum.de/doc/ref.pdf.

[76] S. Peyton-Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. *J. of Functional Programming*, 2(2):127–202, July 1992.

[77] S. Peyton-Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[78] S. Peyton Jones and S. Singh. A tutorial on parallel and concurrent programming in haskell. To appera in LNCS series., 2008.

[79] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.

[80] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000. A preliminary version appeared in *Proceedings, Second Workshop on Higher Order Operational Techniques in Semantics (HOOTS II), Stanford CA, December 1997*, Electronic Notes in Theoretical Computer Science 10, 1998.

[81] A. M. Pitts. Operational semantics and program equivalence. In *Applied Semantics*, volume 2395, pages 378–412. Springer, 2002.

[82] R. Plasmeijer and P. Achten. Generic editors for the world wide web. In *Central-European Functional Programming School – Revised Selected Lectures*, LNCS 4164, pages 1–34, Eötvös Loránd University, Budapest, Hungary, 4–16July 2005. Springer.

[83] R. Plasmeijer and P. Achten. The implementation of iData – a case study in generic programming. In *Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages (IFL 2005)*. Trinity College, University of Dublin, Technical Report TCD-CS-2005-60, 2005.

[84] R. Plasmeijer and M. van Eekelen. *Concurrent* Clean *Language Report (version 2.0)*, December 2001. http://www.cs.kun.nl/~clean/contents/contents.html.

[85] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[86] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[87] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[88] S. Purushothaman and J. Seaman. An adequate operational semantics of sharing in lazy evaluation. Technical Report PSU-CS-91-18, Penn State University, July 1991.

[89] S. Purushothaman and J. Seaman. An adequate operational semantics of sharing in lazy evaluation. In *The 4th European Symposium on Programming (ESOP'92)*, LNCS 582, pages 435–450, Rennes, 1992.

[90] S. Purushothaman and J. Seaman. From operational definitions to abstract semantics. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 276–285, New York, NY, USA, 1993. ACM.

[91] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings, Colloque sur la Programmation, Paris, April 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. 1974.

[92] L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. An operational semantics for distributed lazy evaluation. In *Trends in Functional Programming*, 2009. Chaper 8.

[93] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

[94] D. Sands. From SOS rules to proof principles: An operational metatheory for functional languages. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 428–441. ACM Press, New York, 1997.

[95] J. Seaman and S. Purushothaman. Operational semantics of sharing in lazy evaluation. 1994.

[96] J. Seaman and S. Purushothaman. An operational semantics of sharing in lazy evaluation. *Science of Computer Programming*, 27(3):289–322, 1996.

[97] D. Seidel and J. Voigtländer. Taming selective strictness. In 4. Arbeitstagung Programmiersprachen der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte" im Rahmen der GI-Jahrestagung Informatik 2009, Proceedings, accepted., 2009.

[98] D. Seidel and J. Voigtländer. Taming selective strictness. Technical Report TUD-FI09-06, Technische Universität Dresden, June 2009.

[99] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[100] F. Sinot. Complete laziness: a natural semantics. In Jürgen Giesl, editor, *Proceedings of the 7th International Workshop on Reduction Strategies in Rewriting and Programming (WRS'07)*, volume 204 of *Electronic Notes in Theoretical Computer Science*, pages 129–145, Paris, France, April 2008. Elsevier Science Publishers.

[101] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional programming*, pages 124–132, New York, NY, USA, 2002. ACM Press.

[102] The Coq Development Team. *The* Coq *Proof Assistant Reference Manual, Version 8.2*, 2009. http://coq.inria.fr.

[103] P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton-Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[104] D. Turner. A New Implementation Technique for Applicative Languages. *Software - Practice and Experience*, 9(1):31–49, 1979.

[105] M. van Eekelen and M. de Mol. Reasoning about explicit strictness in a lazy language using mixed lazy/strict semantics. In R Peña, editor, *Proceedings of the 14th International Workshop on the Implementation of Functional Languages, IFL02*, Technical Report 127-02, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, pages 357–373, Madrid, Spain, September 2002.

[106] M. van Eekelen and M. de Mol. Mixed lazy/strict graph semantics. In C. Grelck and F. Huch, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL04*, Technical Report 0408, Christian-Albrechts-Universitt zu Kiel, pages 245–260, Luebeck, Germany, September 2004.

[107] M. van Eekelen and M. de Mol. Proof tool support for explicit strictness. In *IFL*, pages 37–54, 2005.

[108] M. van Eekelen and M. de Mol. *Reflections on Type Theory, λ-calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, chapter Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pages 87–101. Radboud University Nijmegen, 2007.

[109] R. van Kesteren, M. van Eekelen, and M. de Mol. Proof support for generic type classes. In *Trends in Functional Programming*, pages 1–16, 2004.

[110] J. Voigtländer and P. Johann. Selective strictness and parametricity in structural operational semantics. Technical Report TUD-FI06-02, Technische Universität Dresden, June 2006. Revised version accepted to *Theoretical Computer Science*.

[111] J. Voigtländer and P. Johann. Selective strictness and parametricity in structural operational semantics, inequationally. *Theoretical Computer Science*, 388(1–3):290–318, 2007.

[112] D. Vytiniotis, S. Weirich, and S. Peyton Jones. Boxy types: inference for higher-rank types and impredicativity. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 251–262, New York, NY, USA, 2006. ACM.

[113] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, FPCA '89*, pages 347–359. ACM Press, 1989.

[114] C. P. Wadsworth. *Semantics and pragmatics of the lambda calculus*. Ph.D. thesis, University of Oxford, 1971.

[115] M. Wallace. seq vs. pseq, 2006. http://markmail.org/message/d3j5dhzssaxin4od.

[116] N. Yoshida. Optimal reduction in weak lambda-calculus with shared environments. In *Proc. of FPCA'93, Functional Programming and Computer Architecture*, pages 243–252, 1993.

[117] N. Yoshida. Optimal reduction in weak-λ-calculus with shared environments. Technical Report Technical Reprot 93/001, Compurer Science at Keio University, April 1993.